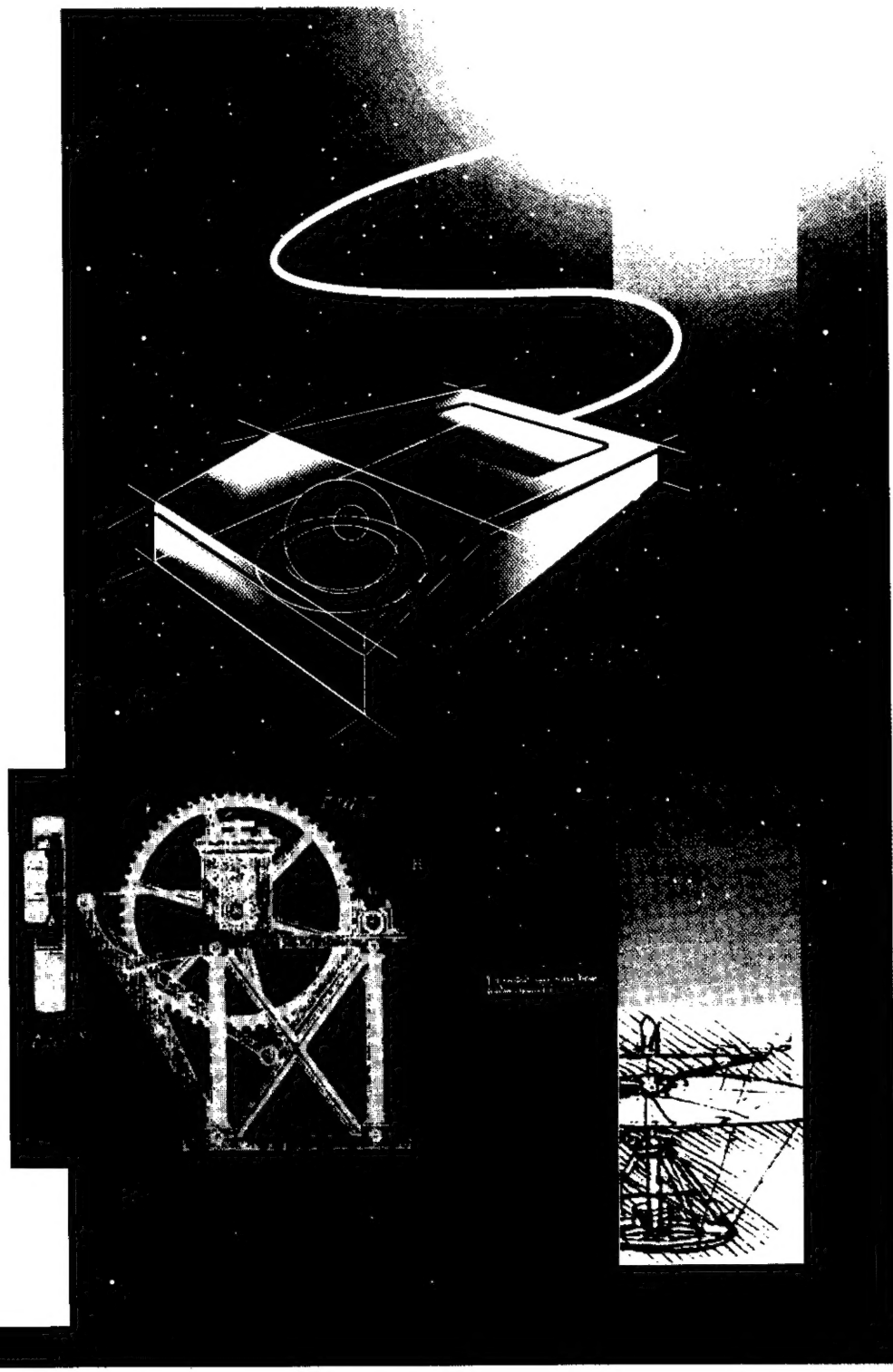
SADE®: Symbolic Application Debugging Environment

Version 1.1

APDA™ # M0614LL/A



SADE® 1.1
Release Notes

This release note describes many of the differences between SADE (the Symbolic Application Debugging Environment) 1.0 and the SADE 1.1 release. This note also documents any known problems with the SADE product.

Reporting Bugs

Please report any bugs you find to Apple. Please use the application "Outside Bug Reporter," found on the MPW Installation Disk. After completing the bug report, send the report:

via AppleLink to: MACDTS

or

via US Mail to:

Apple Computer, Inc.
Developer Technical Support
20525 Mariani Ave. MS 75-3T
Cupertino, CA 95014

SADE Documentation

The following are some places to look for information on how to use SADE:

- The HELP command provides information about the command language.
- The SADE New User Worksheet file contains a variety of commands demonstrating aspects of the command language which you may be likely to use. The Worksheet also contains recipes for debugging MPW tools.and Object Pascal applications
- The SADEStartup file contains the debugger procedure definitions and AddMenu commands implementing SADE's source level debugging interface.

- The :SADEScripts: folder contains some examples of various debugger language constructs. Some of the examples are implementations of builtin SADE commands using the SADE command language. Some information on these scripts is provided in a following section of these notes.
- The SADE Reference Manual.

System Requirements

SADE requires a hard disk and at least 2.5 Mb of RAM. SADE also requires a special version of MultiFinder (supplied) and System 6.0 and Finder 6.1 (or later).

Release Notes

This section contains useful information about SADE that is not contained in the SADE Reference Manual. This release information is divided into the following categories:

- Differences between versions 1.0 and 1.1
- Known Remaining Bugs

Differences Between Versions 1.0 and 1.1

- Source windows are normally opened as readonly when a break point is reached. This prevents you from accidentally modifying your source code in SADE. You can also open source windows read-only through the standard file dialog.
- Source stepping is over 10 times faster than in 1.0. No fooling!
- Pascal WITH statements are supported. You can refer to fields in a WITH statement without qualifying them with the variable when the PC is within the WITH statement.

- Some support for C++ has been added. If you compile your programs with -SYM FULL, C++ will do minimal mangling of field names and local variables, allowing free references to instance variables. References to static member variables is by *TClass::memberName*. «BREAK CLASS *TClass*» and «TRACE CLASS *TClass*» will set break and tracepoints on all member functions of *TClass* . «BREAK OVERLOAD *member*» and «TRACE OVERLOAD *member*» set break and trace points on all overloaded functions whose name is *member*.

- **MACAPP** methods are supported. Within methods, references to the instance variables of the object can be made without qualifying those references by SELF. These free references to instance variables can also be done in C++ member functions.

  To debug Object Pascal and MacApp, set the directory and target your application. Set the Sourcepath for other source files, including the MacApp sources. Open the source file containing your main program and select the statement following the call to INITUMACAPP. Choose the 'GO TIL' command from the SourceCmds menu. When SADE is reentered, execute the following commands:

  ```
  EXECUTE CONCAT(SADEDIR,'SADESCRIPTS:STEPMETHOD.INIT')

  SETUPSTEPMETHOD
  ```

  The following line is in "SADEUserStartup", but commented out. If you are going to debug **MACAPP** a lot, uncomment the line, and the STEPINTOMETHOD debugger procedure will be automatically defined.

  ```
  EXECUTE CONCAT(SADEDIR,'SADESCRIPTS:STEPMETHOD')
  ```

  Use the 'STEP INTO' menu item from the SOURCECMDS menu to step into a method.

  Users of Object Pascal who are not using **MACAPP** should always link with "-OPT ON." Failure to do so will cause SADE to crash.

Known Bugs

This section discusses the known problem areas in the SADE 1.1 release.

- When a breakpoint is set in a tool that hasn't been launched, the targeted tool must be the *next* tool run. SADE does not match tools and their SYM files.

- You cannot debug HyperCard XCMDs or XFCNs with SADE. Hypercard copies the resource into a separate heap block before executing them, preventing SADE from identifying them with symbolic information. This may be fixed in a future version of SADE and/or HyperCard.
- Under some circumstances, SADE may still have a symbol file open while you are trying to update it by rebuilding your application. Unfortunately, there is not a 'NoTarget' command. To close a symbol file, target a different application, either using the 'TARGET' command, or by hitting the SADEKey combination from the alternate application.
- SADE knows only the leaf names of program source files. This means if there are source files in different directories with the same leaf name, the ADDRTOSOURCE function will open the first file it finds, processing the directory paths specified by the SOURCEPATH command, which may or may not be the correct source path. If SADE finds the wrong file, you can change the order the source paths are specified to the SourcePath command, or you can remove the path that contains the errant file.
- Debugging programs whose total number of source lines is greater than 65536 (where did that come from?) does not work properly. As a work around, you can either strip or not generate symbolic information for files you won't be debugging.

  Symbolic information can be stripped with the Lib tool, using the command line

  ```
  Lib -sym off file.c.o -o file.c.o.x
  ```

  or

  ```
  Lib -sym nolines file.c.o -o file.c.o.x
  ```

SADE 1.0 Release Notes

Overview

These notes provide information about how to use the 1.0 release of SADE (the Symbolic Application Debugging Environment).

Reporting Bugs

Please report any bugs you find to Apple. Please use the application "Outside Bug Reporter," found on the MPW Installation Disk. After completing the bug report, send the report to:

via AppleLink: DORIN2, PARR1, and MACDTS

or

via US Mail:

> Apple Computer, Inc.
> Developer Technical Support
> 20525 Mariani Ave. MS 51T
> Cupertino, CA 95014

Where to Look for More Information

The following are some places to look for information on how to use SADE:

- The "Help" command works, providing on-line information about the command language.
- The "New User Worksheet" file contains a variety of commands which demonstrate aspects of the command language which you may be likely to use. The Worksheet also contains a sequence of commands outlining how to debug MPW tools.
- The "SADEStartup" file contains a number of debugger Proc definitions and AddMenu commands which implement a source level debugging interface.

- The "SADEScripts" folder in this release contains some examples of the use of various debugger language constructs. Some of the examples provide implementations of builtin SADE commands using the SADE command language. Some information on these scripts is provided in a following section of these notes.
- The SADE Reference Manual, naturally.

Known Problems In This Release

This 1.0 Final release has the following known bugs:

- When a breakpoint is set in a tool that hasn't been launched, the targeted tool must be the next tool run. SADE does not match tools and their SYM files.
- You cannot debug HyperCard XCMDs or XFCNs with SADE. Hypercard copies the resource into a separate heap block before executing them, preventing SADE from identifying them with symbolic information. This may be fixed in a future version of SADE and/or HyperCard.
- Under some circumstances, SADE may still have a symbol file open while you are trying to update it by rebuilding your application. Unfortunately, there is not a 'NoTarget' command. Once a symbol file is open, the only way to close it is to cause SADE to target to a different application, either using the 'Target' command, or by hitting the SADEKey combination from the alternate application.
- It is easy to redirect output to an open window, and then edit the contents of the window, or even to enter commands from the window. Doing so will have untoward effects–the most likely is that output will be lost or scrambled. This problem exists in the MPW Shell as well, although it is less likely for the user to create the circumstances under which it is encountered.
- SADE only knows the leaf names of program source files. This means if you have source files in different directories with the same leaf name, the AddrToSource function will open the first file it finds, processing the directory paths specified by the SourcePath command, which may or may not be the correct source path. If SADE finds the wrong file, you can change the order the source paths are specified to the SourcePath command, or you can remove the path that contains the errant file.
- Do not launch an application from an application in which breakpoints are set. MultiFinder copies information from the parent to the child application which causes severe problems for SADE in this case.
- Be sure to read the release notes for the appropriate compiler(s) and the linker for the description of several problems with symbol generation.

A Few Helpful Hints

- When you attempt to step into a routine, you may find yourself in a routine that returns a value that is used as a parameter to the routine you really wanted to see. To get where you want to go, first go to the last statement of the currently executing routine. You can do this by selecting the closing brace of the function (in C) or the final end of the function (in Pascal) and using the Go Til command from the SourceCmds menu. Then issue the step into command again.
- When SADE searches for Macsbug names the search is case sensitive. Of course, Macsbug is not case sensitive.
- C has two name spaces: one for the names of structures, and one for all other names. SADE has a single hierarchical name space. This means if you have a structure and a variable with the same name in the same scope, the variable name wins. Note that C++ has a single name space as well, and will report an error at compile time if you attempt this befuddling behavior.
- The Show Value menu item in the Variables menu shows the values of variables. It does not work with expressions. Things like pointer dereferencing and accessing fields in a data structure are expressions, and so the ShowValue command will fail if these are selected. The command is implemented by the proc ShowValue defined in the SADEStartup file. The reason it doesn't work with expressions is that the proc concats the (`) character to the selection, which limits the symbol lookup to the program's scope. This avoids collisions with SADE variables. If you would prefer that the menu item work with expressions at the cost of potentially conflicting with a SADE variable, change the ShowValue proc by removing the "concat('`'," and the matching closing parenthesis from the printf commands.

And a Few Otherwise Undocumented Features

In response to overwhelming user requests, we managed to cram in three new commands: MoveWindow, SizeWindow, and WindowSize. MoveWindow moves the specified window to the specified location. SizeWindow resizes the specified window to the specified size. For both commands, the default window is the target window. WindowSize allows you to specify a rectangle to be used when a window is zoomed, as well as the size and location of a new window. Use the help command to see the syntax of these commands. If you still are confused, MoveWindow and SizeWindow behave as they do in the MPW Shell, and WindowSize sets the ZoomWindowRect and NewWindowRect shell variables.

Differences from SADE 1.0B1

For those who received the Beta 1 version of SADE, these are the major differences between SADE 1.0B1 and SADE 1.0 final:

- You can now click in SADE's windows, set a breakpoint in an application, click in the application's windows, and the breakpoint will be installed. This means you no longer have to have entered SADE from the application via a SADEKey sequence before setting a breakpoint.
- Stepping commands are now nestable. So, if you step over a routine with a breakpoint installed, the breakpoint will be hit. In previous versions, the only execution command allowed in this situation was "Go". You can now step from the breakpoint if you want.
- The size limit for a SADE variable has been increased to about 8K of storage. Of course, a SADE variable can point to larger variables.
- SADE now attempts to display the value of an array of byte-sized values, or a pointer or handle to a byte-size value, as a cstring (displayed in double quotes) or pstring (displayed in single quotes). If the displayed string was found by following a pointer, the quoted string is preceded by a "^" for each level of indirection followed. The first heuristic applied tries to validate the bytes as a cstring by checking for a null-terminated sequence of printable characters. If this fails, SADE then checks for a pstring, treating the first byte as a length byte, and assuring that all of the included bytes are printable. If neither heuristic succeeds, the display will be "value1,..." for an array and "^BYTE(address)" for a pointer, etc. In this case, printable characters means that one of the standard C library routines isgraph() or isspace() returned true.
- SADE now accepts Macsbug names for input. The special character 'µ' is displayed before any Macsbug symbol, and on input, a symbol preceded by the 'µ' operator is looked for as a Macsbug symbol only.
- The Pascal compiler now generates correct symbol information for Object Pascal programs, although it does not generate any symbol information for Objects! The good news is the linker won't crash; the bad news is the resulting symbol file is not very useful.
- The Assembler now outputs source line records, but does not produce type records for RECORD definitions.
- The various layer and menubar problems are fixed.

SADE Scripts

A number of SADE scripts defining procs and funcs (procedures and functions in the debugger's language) have been provided in this release. These procedures and functions can be loaded using the "execute" command on the various script files. The new commands that will then be defined provide a number of higher-level debugging functions which display information about system and program structures. (Some of these were trial implementations of what are now builtin commands in SADE.)

The following files are included:

- DisplayMemory has the definition of a "DM" command to display sections of memory in hex and ASCII. This facility is now built into SADE as the "Dump" command.
- FCBChecker has the definition of a "DisplayFCBs" command which provides a symbolic display of all of the file control blocks in the system.
- HeapProcs has the definitions for "HeapDisplay", "ShowFreeMP", and "DisplayHeapInfo" commands which display the heap, show free master pointers, and display summary information about the heap respectively.
- MiscFunctions has the definitions for "Max" and "Min" which show how to create functions with an arbitrary number of arguments.
- MiscProcs has the definition for "DisplayWindowList" which displays the system window list. For those debugging MPW Tools, there is a proc called "KillTool", which kills an executing tool by calling the MPW Shell routine "SYSRECOVER."
- ResMap has the definition of a "ResMap" command to display a resource map. The builtin "Resource" command performs a similar function.
- ResVerify has the definition of a "ResVerify" command to validate resource maps, similar to the builtin "Resource check" command.
- StackCrawl has the definition of a "StackCrawl" command to display the stack of the program being debugged. This facility is now built into SADE as the "Stack" command.

If you develop any useful and/or fascinating debugger procs and funcs of your own, please send a copy to Apple so that they can be used as manual examples or as part of a library of useful debugging functions.

Contents

Preface **About This Manual**

WELCOME TO SADE™, the Symbolic Application Debugging Environment. Part of the Macintosh® Programmer's Workshop 3.0 (MPW™), SADE is an interactive debugger for programmers writing in high-level languages like Pascal or C.

If you're already writing a program with MPW, you'll have little trouble learning to use SADE. It's a stand-alone application, but the way it runs is very similar to the MPW Shell. As always, you'll need *Inside Macintosh* as a reference.

Part I of this manual introduces SADE's interface, features, and command language.

Chapter 1 gives an overview of SADE. This includes a description of the hardware and software configurations SADE works with, how to install SADE and identify your application, and a tour of the SADE interface.

Chapter 2 describes how to use SADE to debug your application.

Chapter 3 gives a complete description of the various objects used in SADE: how to specify them, and how to create expressions by combining them.

Appendix A contains a summary of all SADE commands.

Appendix B gives examples of program symbols and how to reference them.

Part II provides a complete specification of the SADE command language, including command syntax, operation, and examples. (You can also get the syntax of any SADE command by using the Help command.) ■

Notation conventions

The following notation conventions are used to describe SADE commands:

literal	Plain text indicates a word that must appear in the command exactly as shown. Special symbols (-, §, &, and so on) must also be entered exactly as shown.
variable	Items in italics can be replaced by anything that matches their definition.
[optional]	Square brackets mean that the enclosed elements are optional.
either | or	A vertical bar (|) indicates an either/or choice.
,...	A comma followed by an ellipsis indicates that the preceding item can be repeated one or more times, separated by commas.

Command names are not sensitive to case.

Aids to understanding

Look for these visual cues throughout the manual:

▲ **Warning** Warnings like this indicate potential problems. ▲

△ **Important** Text set off in this manner presents important information. △

◆ *Note:* Text set off in this manner presents notes, reminders, and hints.

For more information

APDA™ provides a wide range of technical products and documentation, from Apple and other suppliers, for programmers and developers who work on Apple equipment. (MPW is distributed through APDA.) For information about APDA, contact the

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 33-G
Cupertino, CA 95014-6299

1-800-282-APDA, or 1-800-282-2732
Fax: 408-562-3971
Telex: 171-576
AppleLink: DEV.CHANNELS

If you plan to develop hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 51-W
Cupertino, CA 95014-6299

Macintosh®

SADE Reference

APPLE COMPUTER, INC.

20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

Part I **SADE Reference**

Chapter 1 **SADE Overview**

THIS CHAPTER INTRODUCES SADE. IT INCLUDES A DESCRIPTION of the hardware and software configurations SADE works with, how to install SADE and identify your application, and a tour of the SADE interface. ■

Contents

About SADE

SADE™ is a debugger for MPW™ programmers working in C and Pascal. SADE is a stand-alone application that runs under MultiFinder™ along with the program you want to debug. If you're not familiar with debuggers, SADE is a program that lets you run the application you've written, stop it, slow it down, and watch it in action.

For instance, if your program just crashes, you can use SADE to see where it was in your code that something went wrong. If you suspect a culprit, you can have SADE interrupt your program when it reaches a certain point. Then you can run one step at a time, or one statement of code at a time, and see what happens. You can watch the values of your program's variables change, or give them different values to see what happens. Or you can look at what values you're passing to toolbox routines.

SADE also lets you examine the state of the system. You can look at the stack to see what procedures have been called. You can display information about the heap and check whether it's in good shape. And you can examine the resource maps of your resources.

These interesting features are all provided by a program that's extremely similar to MPW. SADE is also notable for being a "high-level" debugger. This is not intended as a slight to programmers who like languages like C and Pascal. It simply means that, unlike debuggers that make you understand assembly language and map object code back to your source code, SADE lets you debug in your native language. If you need to look at system things, like actual RAM addresses or what's being put into which register, SADE lets you do that too.

MacsBug users are still included. You can leave MacsBug in the System Folder where it belongs. When SADE is launched, it notifies MultiFinder that it is acting as the debugger for the system. Whenever the System Error Handler is called, or when a 68000 exception occurs, MultiFinder passes control to SADE. Pressing the interrupt switch will still enter MacsBug but it's recommended that you use only one debugger at a time.

Loading SADE

SADE runs on the Macintosh® Plus, Macintosh SE, and Macintosh II computers, and will support future members of the 68000 family. SADE does not run on the Macintosh 128K, Macintosh 512K, Macintosh 512K enhanced, or Macintosh XL.

SADE handles the MC68881 floating-point coprocessor. It does not handle the MC68851 Memory Management Unit (MMU) or the corresponding MMU registers on the MC68030.

Depending on the size of your program, you'll probably want to have at least 2 megabytes of RAM. (It's possible to work with less, but space does get tight.) And if you want to run MPW as well, you'll most likely need a Macintosh with 4 megabytes of RAM.

You may drag the SADE folder from the release disk to anywhere you like on your system; it does not need to be associated with the MPW folder. Note that the SADE application has auxiliary files that should be kept in the same folder as SADE: SADEStartup, SADEUserStartup, SADE.Help, SADE Worksheet, and SADE New User Worksheet.

The file SysErrs.Err is also provided and should be kept in either the SADE folder or in the System Folder so that SADE can report system-related errors. The SADEScripts folder contains examples and can be kept anywhere or omitted altogether.

The SADE release disk also includes a special version of MultiFinder, compatible with System 6.0, that's used to control and access processes. The additional code to support debugging will not affect your normal use of MultiFinder. To install this special version, move your current MultiFinder out of the System Folder, copy the new version in, and reboot. If you already have version 6.1 or greater of MultiFinder, you don't need to bother with this step.

Getting started

To launch SADE, double-click the SADE application icon or a SADE document icon. If you have MPW running, you can also launch SADE from the Shell.

The SADE interface is extremely similar to that of MPW. The File, Edit, Find, Mark, and Window menus are almost identical to those in MPW 3.0. SADE has two additional menus that provide the basic source-level debugging functions.

Once launched, SADE opens the SADE Worksheet file. As in MPW, you can enter SADE commands from the SADE Worksheet, using the standard MPW editing, selecting, and executing conventions.

To get the syntax of any command, type `help` and press Enter. To see some example commands and how to use them, type `open 'SADE New User Worksheet'` and press Enter. To find a complete description of every single command, look at Part II of this manual.

Entering commands

A SADE command line is very similar to an MPW command line, but there's one major difference. SADE provides full expression evaluation. In order to distinguish strings from other types of expressions, you need to enclose them in quotation marks. For instance, to open a file, type `open 'filename'`. Pathnames, directory names, filenames, menu names—all must be enclosed by quotation marks.

You can enter multiple commands on the same line, separated by semicolons (;). If you want to continue a line, precede the carriage return with the escape character (∂). You can put comments anywhere in the command line, except within strings. Comments are delimited by a number sign (#) and by the end of the line.

Command output, also known as standard output, is displayed just below the command that was executed. If you want, you can direct output to a different window, or file, with the Redirect command. For instance, enter the following three lines on the SADE Worksheet:

```
open 'new'          #remember the quotes
redirect 'new'
'hello'; version; help version
```

You should see SADE create a window called New, and display the string "hello" and the SADE version information in the new window.

For SADE's purposes, a window and a file are synonymous. All SADE windows are opened as text files.

To undo redirection, simply type `redirect` without parameters.

- *Note*: If a run-time error occurs while SADE commands are executing, the commands are aborted, and error messages are written to the command window; that is, the window from which you enter the command. If output was redirected to another window, that redirection is undone.

Command output can be as complex as the application you're debugging. The Printf command gives you the ability to convert, format, and display almost any type of value in whatever way you like.

Identifying your program to SADE

For SADE to work, it needs to have symbolic information about your application. When you compile and link your program, use the `-sym on` option; this generates a file called "YourProgName.SYM." The linker puts this file in the same directory as your application.

Your files can be anywhere on the system, but you need to tell SADE where they are. The Directory command specifies the default directory for all SADE file-oriented commands; using it relieves you from having to specify full pathnames all the time:

```
directory 'VolName:path:to:MyAppDir'
```

The SourcePath command tells SADE which directory your source files are in; if they're in the default directory, you don't need to use SourcePath. If your source files are in another directory, or if you want to keep them in more than one directory, use SourcePath:

```
sourcepath 'VolName:path:to:MySourcesDir'
```

The Target command tells SADE which application you want to debug:

```
target 'MyAppName'
```

The next chapter describes how to use SADE to debug your program.

Chapter 2 **Debugging With SADE**

THIS CHAPTER DESCRIBES HOW TO USE SADE's menus and commands in the context of typical debugging situations. ■

Contents

Starting and stopping

You can launch your application from the Finder, or from SADE by using the Launch command: `launch 'MyApp'`. When both SADE and your application are running, bring your application up to the front and press the key combination "Command-Option-." using the period key on the numeric keypad. (Don't press the interrupt switch unless you want to enter MacsBug.) This key combination is known as the SADEKey.

If you don't like the Command-Option-. key combination, you can define another Command-Option key combination with the SADEKey command.

When the SADEKey is pressed, MultiFinder calls SADE the next time your application calls the WaitNextEvent, GetNextEvent, or EventAvail routines. SADE looks at the stack for the location from which you called the event trap, sets a breakpoint at the next instruction, and restarts your program. As soon as the event trap returns, the breakpoint is hit, and SADE takes over.

◆ *Note:* If your program happens to be stuck in a loop from which it does not call one of the event traps, you'll need to press the SADEKey a second time. At this point, SADE is entered immediately. In most cases, the state of the system is reliable, though occasionally the system may be in a state in which SADE cannot run.

SADE opens your source file in front as a read-only window, highlights the statement from which the event call was made, and displays a dialog box describing the nature and location of the exception.

▲ **Warning** If you are running both SADE and MPW, it's important that, when in SADE, you always open your source files read-only. ▲

◆ *Note:* You'll notice that SADE sometimes selects several source statements as if they were a single statement. This occurs with statements for which no code is generated, resulting in several statements that map to the same address.

If SADE can't find your source files, or if you didn't use the `-sym` option when you compiled and linked your program, it displays a message on the SADE Worksheet, including a disassembly of the instruction at the program counter. SADE also displays such a message when the routine that was executing isn't part of your program (ROM code, for instance).

To set your program running again, choose Go from the SourceCmds menu.

More times than not, you'll want to suspend your program at a particular place by setting a breakpoint. You can set a breakpoint either while your program is interrupted or before it is launched. To set a breakpoint, just click in the source statement you want to stop at and choose Break from the SourceCmds menu. The next time your program runs, SADE interrupts it just before the statement with the breakpoint is executed.

Another way to interrupt your program is to have it call the toolbox routine SysErr with an unused system error ID in the range 129 to 32,511.

The Break If menu item lets you specify a condition for breaking; for instance, when a variable has a particular value.

- *Note:* In a C function, to set a breakpoint that's hit each time through a While loop, put the cursor in the condition part of the While statement (rather than in the word "While") and choose Break from the SourceCmds menu. (The C compiler generates two statements for a While statement—one that executes the first time only and another that executes each time the controlling condition is evaluated.)

Breakpoints remain set after you resume execution; to remove a breakpoint, choose the Unbreak from the SourceCmds menu while the statement with the break is selected.

After your program has been suspended, you can move around SADE as you like. You may want to move to the SADE Worksheet window and execute a command. For instance, if you want to see what routine called the procedure that was interrupted, you can use the Stack command. Stack displays a list of current stack frames, letting you examine the procedure calling chain.

If you want to get back to the source statement at which your program was interrupted, you can choose In What Statement from the SourceCmds menu. This menu item displays and highlights the statement that corresponds to the instruction in the program counter; it's handy for bringing back the source file and finding the current execution point.

The Statement Selected Is? menu item displays a dialog box giving the procedure name and offset of the statement that's selected in the active window. You can use this information when entering commands (to set a breakpoint, for instance) from the Worksheet.

The Show Selected Routine menu item displays the source for the routine whose name is selected. You can use this menu item in conjunction with the Stack command to see the source for a particular point in the calling chain.

If your program is hopelessly fouled up, you can use the Kill command to terminate it. You can also use Kill to terminate a perfectly healthy application. Be aware, though, that Kill is dangerous, because it doesn't give the application a chance to perform its usual exit routines (like saving data). Kill does perform some cleanup activities like freeing the MultiFinder memory occupied by the application and removing its trap patches.

To quit SADE, use the Quit command. To shut down the system from within SADE, use the Shutdown command; it terminates SADE and calls the Shutdown Manager. Shutdown lets you specify a restart as well.

Controlling program execution

Breakpoints require several operations: You need to set them, go, and then remove them. If your program is interrupted and you just want to move a bit further, select the statement you want to stop at and choose Go Til from the SourceCmds menu.

You can also step through your program, watching statements execute one at a time. There are two ways of stepping, depending on whether you're interested in stepping through subroutines when they're encountered. If you want SADE to step over procedure calls (treating BSRs and JSRs as single instructions), choose Step from the SourceCmds menu. Step executes the statement that's selected in the source window, suspends execution, and highlights the next statement to be executed.

If you want to step into subroutines as well, choose Step Into. SADE then steps in, stopping at the first instruction of the called procedure. If you choose Step Into and then find you don't want to be there, choose Step Out. This menu item makes SADE move through the subroutine and stop at the first statement following its return.

Toolbox traps are always treated as single instructions; SADE steps over them, stopping at the first instruction following the trap.

You will also probably want to watch the values of your program's variables. For instance, you may suspect that a loop using a count variable has a problem. There are two ways of looking at variables. If you select a variable and choose Show Value from the Variables menu, SADE creates a file called "Values" and puts the current value of that variable in it. If you choose Show Value for another variable, its value is added to the file. To have a value displayed in hexadecimal, choose Show Value in Hex.

The "Values" file isn't updated to reflect changes in the value of a variable as you continue to run your program. If you want to look at the different values a variable takes on, select the variable and choose Add Watch Variable. SADE creates a different file, called "Variable Watch." Each time control returns to SADE, the file is updated to show the latest values.

To remove a watch variable, select the variable and choose Delete Watch Variable. To delete all watch variables, simply choose the Delete All Watch Variables menu item. (Note that the information in the "Variable Watch" file is preserved.)

The final SourceCmds menu item, Source [vs. Asm] Debugging lets you use the Step, Step Into, and Step Out menu items at the assembly-language level as well. When you choose assembly-language debugging, Step and Step Into execute a single assembly-language instruction at a time.

- *Note:* The SourceCmds and Variables menu are implemented in the SADEStartup file and can be modified to suit your needs. For details, see the section "The SADEStartup File" later in this chapter.

More SADE commands

All of the operations implemented by the SourceCmds and Variables menus can also be performed by entering SADE commands. When working with source files, you'll probably find it easier to use the mouse and the menus. In some cases, though, the equivalent command provides further options. And there are many additional SADE commands that perform functions not provided by the menus.

For instance, the Break command lets you set multiple breakpoints with a single command, and you can set breakpoints on traps as well as on program statements. Break also lets you specify a break action to be performed when the breakpoint is reached. (Break actions are described below.) To remove breakpoints, use the Unbreak command.

If you want to monitor addresses or traps without suspending your program, you can use the Trace command. After setting a tracepoint, resume execution by executing the Go command (or by choosing the Go menu item). When the tracepoint is reached, a message is written to standard output reporting the address or trap being traced. To remove tracepoints, use the Untrace command.

If you want to see what breakpoints and tracepoints are set, you can use the List command to display a list. You can also use List to see what MultiFinder processes are active.

Debugging often involves going below the symbolic level and looking at the state of the system. You may, for instance, need to look at the contents of a specific RAM address or range; you can do this with the Dump command. To find a value or string in memory, use the Find command.

If you're familiar with assembly language, it's sometimes useful to look at your program, or at code other than your own, from the object-code level. You can use the Disasm command to disassemble assembly-language instructions. You might use Disasm to investigate how your source statements map into object code. Or you could use Disasm to look closely at how parameters are passed to toolbox routines.

Three commands let you examine and check the heap. The Heap command displays information about the current heap. By examining the different types of blocks and how they're allocated, you can detect problems like heap fragmentation. You can use the Heap Check command to verify the consistency of the heap if you think it might be corrupted. Finally, the Heap Totals command gives you a summary of the heap allocation.

For looking at resources, the Resource command displays resource maps and the Resource Check command checks them for consistency.

SADE lets you define macros to use in place of just about anything you might type on the command line. You can define a simpler name for a command, or create a macro for a long string of characters. To simplify the specification of a long directory path, for instance, you could enter the following commands:

```
macro dir 'directory'
macro path 'VolName:very:long:path:to:default:directory'
dir path    #quick way to specify default directory
```

Programming in SADE

The real power of SADE is that, in addition to using commands interactively, you can write your own programs, using SADE commands, to automate repetitive tasks, perform complex debugging operations, or even customize the SADE interface itself.

The simplest SADE program is a group of commands enclosed by the Begin...End construct. One common use of this construct is with the Break command to group a sequence of commands to be performed when the breakpoint is encountered.

Break actions

A break action can be a single command, a procedure or function, or a group of commands delimited by the Begin...End construct.

If you don't specify a break action, SADE simply stops when the breakpoint is reached. When you do specify a break action, SADE resumes execution after performing the action.

You can suspend execution as part of a break action by using either the Abort or the Stop commands. The Stop command suspends execution, but first executes any pending commands. That is, if the most recent execution statement was in a structured statement, or if multiple commands were selected, these commands are executed before you enter SADE. The Abort command suspends execution, but cancels any pending commands.

The Alert command displays a message; you might use Alert as part of a break action.

SADE procedures and functions

You can write SADE procedures and functions using familiar programming constructs like If...Else...End and While...End. You can also define SADE variables with the Define command.

SADE provides a set of built-in functions, as well as a number of predefined variables; they're described in Chapter 3. The SADEScripts folder contains many examples of useful procedures and functions.

To use a procedure or function, you first need to execute the file containing the routine by using the Execute command. Once the script has been executed, you can invoke the routines defined within it by name.

For instance, the procedure `displayWindowList`, used below as an example, is contained in the script MiscProcs (found in the SADEScripts folder). To use this procedure, you first need to execute the script:

```
execute 'VolName:path:to:SADEScripts:MiscProcs'
```

Procedures are delimited by the Proc...End construct, and functions are delimited by the Func...End construct.

The If...Else...End construct lets you specify conditional execution of a number of different actions. You can loop with a conditional test at the beginning of the loop with the While...End construct, or with a test at the end of the loop with the Repeat...Until construct. Or you can use the Loop command with the Leave command providing a conditional exit wherever you like. The Leave command can be used with any of the other looping commands as well.

The `displayWindowList` procedure displays the window titles for all the current windows:

```
proc displayWindowList;
define awindow;
awindow := ^WindowRecord(windowlist);
while (awindow <> 0) do
  printf ("Window Title = \"%P\"\n",^pString(awindow^.titleHandle^)^);
  awindow := ^WindowRecord(awindow^.nextWindow);
end;
end;
```

The Define command defines a variable called `awindow`. SADE variables are dynamically typed; that is, their type is determined on assignment (and may be changed by new assignments). `Awindow` is used here to contain a pointer to a window record.

A pointer to the window list is maintained in the low-memory global variable Windowlist. The pointer to the window record for the first window in the list is assigned to `awindow`. The commands within the While...End construct are executed as long as `awindow` does not equal NIL. The window list is null-terminated, so when the entire list has been processed, the While condition is no longer true and the procedure returns. Each time through the While loop, the Printf command prints out the title (using the titleHandle field of the window record) and the next window record pointer is assigned to `awindow`.

The Return command returns from a procedure or function. Return is optional for procedures, but functions must use a Return command to pass the function result.

To remove the definition of a SADE variable, procedure, function, or macro, use the Undefine command.

The SADEStartup file

Each time you launch SADE, it executes the SADEStartup file. This file contains SADE commands, functions, procedures, and variable definitions that configure SADE for source-level debugging. In particular, the SADEStartup file creates the SourceCmds and Variables menus. If you study the way each of the features is implemented, you can change them to suit your own needs.

For instance, when you interrupt your program, SADE brings the source window up as the frontmost window, and displays an alert. These actions are controlled by definitions in SADEStartup:

```
define SourceInFront := 1      #source brought up as frontmost window
define BreakAlert    := 1      #display an alert at break
```

You can change the values of these variables from the SADE Worksheet if you like by entering:

```
SourceInFront := 0             #source brought up behind command window
BreakAlert    := 0             #don't display an alert at break
```

The new values remain in effect for that debugging session only. The next time SADE is launched, the definitions in SADEStartup are executed again, restoring the default display.

If you want to redefine a certain feature indefinitely, or to add new features, you can put new definitions into the SADEUserStartup file. Initially empty, SADEUserStartup is executed by SADEStartup. You can change the definitions in SADEStartup directly, but it's safer to change SADEUserStartup so you don't accidentally edit something useful in SADEStartup.

You can enter SADE commands and define your own procedures and functions in SADEUserStartup. For instance, you can create new menus for executing commands by using AddMenu commands.

If you have routines in other files, you can execute the file from the SADE Worksheet, or you can put the Execute command in SADEUserStartup and have the file executed automatically at startup.

The OnEntry command specifies what to do each time SADE is entered. OnEntry can take a single command, a group of commands, a procedure, or a function as an argument.

In SADEStartup, the procedure `StandardEntry` is passed to the OnEntry command. `StandardEntry` performs a number of useful actions. It identifies where your program code is interrupted. It also provides for source display of the current program counter (PC) on entry into SADE. If the source cannot be displayed, it executes a Printf command that shows the cause of the interruption, the location of the program counter at the time the error occurred, and the name of the program that was halted. The numeric codes listed in this procedure correspond to the different types of interrupts and system errors that could occur.

You can use the OnEntry command to specify additional or different actions, but you need to know what you're doing. Each time the OnEntry command is entered, the actions specified by the previous OnEntry command are replaced. You'll probably want to use the `StandardEntry` procedure as a model, modify it to suit your needs, and pass it to OnEntry in the SADEUserStartup file.

The "Show Value" and "Add Watch Variable" menu items create files in the current directory and redirect output to them. If you prefer to have these files created in a different directory, you can modify the procedures that implement the menu items. For instance, you could modify the Open command in the `ShowValue` procedure:

```
open behind "VolName:path:to:values"
```

A set of Macro commands define macros with MacsBug-like syntax for certain operations. For instance, the `td` macro executes the `DisplayRegs` procedure. `DisplayRegs` opens a window called "register display" and displays the current values of registers D0–D7, A0–A7, and the program counter. (The Open command in this procedure can also be modified if you want the "register display" file created in a different directory.)

The `setSourceBreak`, `unSetSourceBreak`, and `sourceStep` procedures support source-level breakpoints, allowing you to identify code locations by pointing at your program source. You probably won't need to modify the behavior of these procedures.

Chapter 3 **Symbols, Constants, and Expressions**

THIS CHAPTER GIVES A COMPLETE DESCRIPTION of the various objects used in SADE: how to specify them, and how to create expressions by combining them. ■

Contents

About symbols

This chapter discusses everything you could possibly put on the command line, and describes the rules SADE uses in interpreting this information.

To display the value of a symbol, simply enter the name. SADE evaluates the symbol and automatically displays it according to its type. The default radix for numeric types is decimal. Address values (pointers) are displayed in hexadecimal.

In a SADE debugging session, a variety of different symbols are used. Your program has compilation units that contain procedures and functions, as well as global and local variables. There are Macintosh system symbols like toolbox traps, registers, and low-memory global variables. Finally, SADE itself defines, and lets you define, commands, procedures, functions, macros, and variables.

When executing the command line, SADE takes the first symbol and tries to identify it first as a debugger symbol—a command, a procedure or function, or a variable. It then tries to identify it as a program symbol, and finally as a system symbol.

It's possible for symbols in different classes to have the same name. A reference to a system symbol, for instance, could be masked out if SADE finds a debugger or program symbol with the same name. For this reason, SADE provides two operators to specify program and system symbols:

- The backquote character (`) indicates a program symbol.
- The delta character (Δ) indicates a system symbol.

For example, if you have a program symbol whose name is `pc` just like the system symbol for the program counter, you could use `Δpc` to refer to the system symbol. In addition to avoiding masking, these operators speed up the search process.

Another operator, the μ (Option-m) character, indicates a MacsBug symbol that's embedded in a program object. You can use this operator to specify a MacsBug symbol for a module with no symbol information, without slowing the search process. Conversely, whenever SADE displays a MacsBug symbol, it precedes the symbol with this character.

The first character of a symbol identifier must be an uppercase or lowercase letter (A-Z, a-z), an underscore (_), or a percent sign (%). Subsequent characters can be letters, digits (0-9), underscores (_), dollar signs ($), number signs (#), percent signs (%), or "at" symbols (@). Other characters may be made a part of an identifier by preceding them with an escape symbol (∂) or backslash character (\) and enclosing them in double quotes. A name may be any length, but only the first 63 characters are significant.

When looking up symbols, SADE first performs a lookup being sensitive to case. If the symbol isn't found, SADE converts all the characters to uppercase and looks again. Pascal programmers may want to use the Case command to turn case sensitivity off; SADE will then convert all symbols to uppercase before any lookup, speeding up the search process.

A symbol is meaningful only within the scope for which it is defined. For instance, local variable references are meaningful only within the procedure or function in which they are defined. SADE can't interpret a symbol until a memory location has been allocated for it. If a program is interrupted before a procedure containing symbol definitions has a chance to execute, SADE can't find the symbols within that procedure.

A symbol's memory location may also become inaccessible during execution. For example, Pascal and C will place a procedure's variables into registers. If one procedure calls another, the calling procedure's variables are no longer available.

Debugger and system symbols exist in flat namespaces. Program symbols, on the other hand, exist in a hierarchical namespace that begins with the level of your program as a whole, and can end with the fields of complex data structures defined within nested procedures. For this reason, it's important to understand and be aware of how program symbols are specified. (Appendix B gives examples, in both C and Pascal, of how to specify program symbols.)

Program symbols

It's possible to give a fully qualified reference for any program symbol that starts with the program's global scope and continues down the hierarchy through the name of the particular symbol:

```
\unit [.procedure]* .variable
```

The components of such a reference are as follows:

\	The backslash character is the program-level, or unit qualifier, and should precede the unit name.
unit	When referring to the main program's variables, the unit name is the program name. For Pascal, it's the given unit name or the name on the PROGRAM statement if it's the main program unit. For assembly language or C, it's the name of the compilation unit; in other words, the filename without the extension (.c or .a). Special characters in the filename must be quoted with the escape character (∂).

[...]* The [...]* construct indicates that zero or more levels of procedure name qualification may be used: zero when accessing unit level variables, one when accessing first/level procedures, and more when accessing nested procedures. (This is possible in Pascal but not in C.)

The period character (.) is the procedure and variable name delimiter.

procedure A procedure (or function) reference refers to the starting code location of procedures (or functions). Among other things, a procedure reference may be used in setting breakpoints.

variable This can be constructed according to the rules for simple and structured variable references, as described below.

A reference that begins with the program-level qualifier (the "\" backslash character) is essentially fully qualified. SADE treats the identifier following the backslash as a global, and runs through the remaining qualifications. To specify correctly a procedure starting at the compilation unit level, you must precede the compilation unit name with a backslash. Otherwise, SADE can't determine whether the name refers to a compilation unit, a procedure, or a variable.

SADE allows partially qualified references if the omitted information can be deduced from the current execution point. That is, if program execution is currently suspended at some point, that point identifies a program, compilation unit, and perhaps procedure from which to begin looking up variable references.

A procedure reference refers to the starting code location of a procedure (or function). Procedure references are used, for instance, in setting breakpoints. The first, or 0th, statement of a procedure or function corresponds to the entry point to that routine before the LINK instruction has been executed. At this time, the stack frame has not been set up and the local variables and parameters do not yet have meaningful values. If you wish to check the value of a parameter you should go to statement 1, in other words `foo.(1)` instead of `foo` or `foo.(0)`, before checking the value.

Procedure and function names, as well as statement references relative to procedures, can be used as arguments to commands, such as in `break foo`. If you wish to assign the address of a procedure to a register, you should precede the name with the @ operator, as in `pc := @foo`. In the former case, the name or statement reference is a special code reference that includes information about resources and offsets, while the latter case is a simple address at a specific point in time.

Source program statements are identified by indices relative to a procedure. These indices are identified by the compiler and are associated with locations in the text in source windows. A statement reference consists of a procedure reference followed by a reference of the form `.(expr)`, which refers to a particular statement index relative to the specified procedure. If the procedure reference is omitted and the `.(expr)` form is used by itself, it is taken to refer to the current procedure at the time execution was suspended.

Reference to symbols outside the current stack activation is supported by an array-like specification. An expression within square brackets is inserted between the last procedure name and the beginning of a variable reference, to indicate the *n*th activation of that procedure from the top of the stack. For instance `myproc[3].varname` refers to the variable vamame belonging to the third most recursive call to the procedure myproc.

A simple variable reference consists of a single identifier, such as `myVar`, and can be of any type supported in your program. SADE also supports structured types common in high-level programming languages, using the following operators:

```
.name or    ->name        record or structure member selection
^    or     *             pointer dereference
[n,...]  or   [n]...      array access
```

Variable references can be as complex as necessary (keeping in mind, of course, that each type supports the operator applied to it). For instance, `myRecord.myArrayVar[1]` references the first element of the array myArrayVar which is an element of the structure myRecord. Note that in SADE, array variables are 1-based.

In SADE, a variable reference always refers to its value, and not its address. For instance, `foo` refers to the value of the variable "foo" and `@foo` (or `&foo`, in C) refers to its address.

Symbolic information from the Pascal compiler does not include information about the use of WITH statements so SADE cannot deal with unqualified field references. To access the fields of a record that has been identified with a WITH statement, you need to include the name of the record variable (as you would were the WITH statement not present).

Variables that are defined as extern in a C source file, but whose defining file was not compiled with the `-sym` option, are not known to SADE.

Predefined SADE variables

A number of predefined SADE variables provide access to state information, such as exceptions, date, and process ID. Many of these variables are read-only and cannot be assigned values. The predefined SADE variables are as follows:

ActiveWindow A string containing the name of the frontmost (active) SADE window. This is a read-only variable.

Arg [*n*] The *n* th parameter of the current SADE procedure. This variable is used like an array variable to access the parameters of the current SADE procedure numerically rather than by name. Note that this array variable is 1-based.

Date A string containing the the current date in the form "dd-mm-yy." This is a read-only variable.

DisAsmFormat A string used to format the output of the Disasm command. Each line of the output is divided into four fields. The presence, order, and format of the four fields are controlled by "flag" letters in this variable. The initial value is OAXC, which specifies offset, address, hex representation, and finally the assembly code. You can change the output by using the following flags:

o	Display offset field in decimal.
O	Display offset field in hexadecimal.
a, A	Display the address (case is not significant).
x, X	Display the hex code representation (case is not significant)
c	Truncate the assembly code if necessary to a uniform length
C	Show entire assembly code no matter how long.
$	Prefix offset and/or address with a $ (allowed only before O, a, and A flags).

Blanks and tabs are ignored in the string. A flag specifying the presence of a field may not be repeated. At least one of the two flags x/X or c/C must be specified. If the assembly code field is specified as the last field, then c has the same meaning as C—the entire assembly code field is displayed. If the assembly code field is to be displayed before one of the other fields, you then have the option of either truncating the assembly field to a uniform length (c) or showing it completely (C).

The DisAsmFormat variable may also contain a $ flag in front of the O, a, or A flags to generate a $ character in front of the offset and/or address field values.

Exception The exception number of the most recently encountered exception. This is a read-only variable. Possible values include the standard system error IDs (listed in the MPW file SysErr.a), error IDs that you define, as well as the following special SADE exception numbers:

50 A-trap break
51 Instruction trace
55 User interrupt (SADEKey pressed)
58 Address break
63 Nonfatal internal error
64 Fatal internal error

Inf Always equal to a SANE infinity. Inf is a read-only variable.

NArgs The number of actual parameters specified for the current SADE procedure or function. NArgs is undefined when no procedure is in use. NArgs is a read-only variable.

ProcessId The process identifier for the current target program. It identifies the process that was suspended when SADE was entered. ProcessID is updated when the Target command is executed. (ProcessID has a negative value if the target has not been launched.) This is a read-only variable.

TargetWindow A string containing the pathname of the file open just behind the current window. This is a read-only variable.

WorksheetWindow A string containing the pathname of the SADE Worksheet window. This is a read-only variable.

Register names

Register names are system symbols. SADE uses these names to display the data your program places into the registers provided by the 68000 family of microprocessors. When you disassemble instructions, you can see what registers were used by a particular instruction.

These register names may be used from SADE:

D0...D7	Data registers
A0...A7	Address registers
CCR	Condition code register
SR	Status register
SP	Stack pointer
PC	Program counter
FPCR	Floating-point control register
FPSR	Floating-point status register
FPIAR	Floating-point instruction address register
FP0...FP7	Floating-point data registers

Expressions

Expressions are composed of either a single term or an arithmetic combination of terms. A term is either a named symbol, a constant, or a function call. Terms are combined by arithmetic, logical, shift, and relational operators. String terms may be combined only with relational operators or string functions.

Numeric constants

Numeric constants take the form of decimal, hexadecimal, binary, and floating-point numbers.

Decimal: Decimal numbers are formed as a string of decimal digits (0–9). Values are treated as 32-bit (signed long word) quantities. Decimal values that exceed 32 bits are treated as floating-point values. To enter an unsigned value, you must coerce the right side:

```
x := UnsignedLong(value)
```

Hexadecimal: Hexadecimal numbers are specified by a dollar sign ($) followed by a sequence of hexadecimal digits (0–9, A–F, or a–f). Hexadecimal numbers are treated as 32-bit quantities and are left-padded with zeros if necessary; in other words, $FF is treated as $000000FF. For convenience, you can use periods to separate digits.

Binary: Binary numbers are specified by a percent sign (%) followed by a sequence of binary digits (0–1). Binary numbers are treated as 32-bit quantities and are left-padded with zeros if necessary. For convenience, you can use periods to separate digits.

Floating-point: Floating-point numbers are specified with a decimal point or exponent as described in the *Apple Numerics Manual*. Within SADE, floating-point numbers are represented as SANE 10-byte extended values.

Hexadecimal and binary numbers that are longer than 32-bits are treated as strings; this is useful for arbitrary assignment of values. (If these strings are used in an expression, they're treated as SANE extended numbers.)

Strings

Many of the SADE commands take a string as a parameter. A string is defined as a sequence of one or more ASCII characters (including blanks) enclosed in single (') or double (") quotation marks. Strings are limited to a length of 254 characters.

Escaped characters may be specified in double-quoted strings. An escaped character is represented by an escape character (∂) or backslash (\) immediately followed by one to three decimal digits, or by a 1- or 2-digit hexadecimal number (\$xx), or by one of the following single character reserved to represent certain nongraphic characters: `\n` (newline, \$0D); `\t` (tab, \$09); and `\f` (formfeed, \$0C). Any other character immediately following the backslash represents just that character, for example, `\\` (backslash); `\'` (single quote); and so on. You can also delineate the string with single quotation marks and use the double quotation marks as literals, or vice versa. Note that this special character processing does not occur when single quotes are used as delimiters.

Here are some examples of strings:

```
'Hello'          'don''t'          "hello\0"  (null-terminated)
"Hello"          "don't"           "don""t"
"'"              ''''  (single quote)   """"  (one double quote)
```

String constants used in arithmetic expressions are limited to four characters. Such strings are treated as right-justified 32-bit signed values. Each of the characters in such a string is assigned its ASCII value, and the overall value of the string represents the concatenation of values of its elements. For instance, the string "my" has the value: $6D * 256 + $79.

- *Note:* When executing a file, each command line is limited to a maximum of 254 characters. A maximum-length string of 254 characters is too long when executed from a file, because the quotation marks used as delimiters are counted as part of the line length.

SADE treats C strings, which are null-terminated arrays of characters, as arrays of unsigned bytes. It does, however, attempt to display arrays of 1-byte values, as well as pointers and handles to such values, as either C or Pascal strings.

Built-in SADE functions

SADE provides a set of built-in functions that perform useful operations. Each of these built-in functions may be used as part of an expression.

`AddrToSource (`*address* `[,` *Boolean* `] )`

AddrToSource displays and selects the source statement corresponding to the specified address. If the source file isn't already open, it's brought up as a read-only window. If the source file is already open as a read/write window, AddrToSource changes the window to read-only.

If the optional *Boolean* is omitted or is false, the source window is displayed behind the frontmost window (from which AddrToSource was likely to have issued). If the *Boolean* is nonzero, the window is brought up as the frontmost window.

AddrToSource returns a Boolean value indicating whether it was able to display the source (true) or not (false).

`Concat ( [` *string,...* `] )`

Concat returns the concatenation of the specified string expressions. If given nonstring arguments, Concat tries to coerce them to strings. If no arguments are specified, Concat returns a null string.

`Confirm (` *message* `[,` *Boolean* `] )`

Confirm presents a dialog box containing the specified message and returns a number indicating the response. When the optional Boolean is omitted or is false, Confirm presents OK and Cancel buttons, and returns 1 or 0 respectively. If the Boolean is nonzero, Yes, No, and Cancel buttons return 1, 0, and –1 respectively.

`Copy ( string, characterIndex, length )`

Copy can be used to copy all or part of the specified string. *CharacterIndex* specifies the first character to copy; to start with the third character of the string, for instance, you would specify 3 (it's 1-based). The length of the substring is determined by the optional *length* or by the end of the string.

`Eval (text, [ message ] )`

Eval evaluates the text of a string argument as an expression. The function result can be of any type, depending on what the expression evaluates to. You can optionally supply a message; if an error occurs, this message is returned as the function result. If no message is supplied and an error occurs, Eval is aborted and the error is reported.

`Find ( target, address, length [,count ] )`

Find looks for a target pattern in the memory range specified by a starting *address* and *length*. If you pass zero for *count*, Find returns the number of occurrences of the pattern. You can use the *count* parameter to specify which occurrence of the pattern you want; for instance, if you specify 3 for *count*, Find returns the address of the third occurrence of the pattern. If you omit *count*, Find returns the address of the first occurrence. If the target is not found, Find returns 0.

Remember that expression values in SADE are long words by default; to specify another size, use a type coercion. For instance, `Find($ABCD,PC,20)` looks for a long value, and `Find(word($ABCD),PC,20)` looks for a word value.

`Length ( string )`

Length returns the length in bytes of the specified string.

`NaN ( expression )`

The NaN function converts the specified expression into a SANE 10-byte extended value.

`Request ( string [, string ] )`

Request returns a string after displaying a request dialog box. The first string argument is displayed in the dialog box as the request message. The second, optional string argument specifies a default string to present in the request box.

When you click Cancel, the string "_CANCEL_" is returned. Otherwise, the string specified is returned.

`Selection ( windowName )`

Selection returns the text of the current selection in the specified window. The value returned is of type PString. To get the value of the string if it contains a name or expression (for instance, to set a breakpoint), apply the Eval function on the string.

`SizeOf ( variable | type | argument )`

SizeOf returns the number of bytes occupied by a variable or type. You can also use SizeOf to determine the size of an argument to a SADE procedure or function. SizeOf cannot be used with SADE array variables.

`SourceToAddr ( windowName [, errorFlag ] )`

SourceToAddr function returns the address corresponding to the statement selected in the specified window. The window need not be active. The address is displayed symbolically; if the address cannot be determined, SourceToAddr returns zero.

If you pass a nonzero value in the optional *errorFlag*, SourceToAddr returns a string describing why the address could not be found.

Example:

```
foo := SourceToAddr(targetWindow,1)
if TypeOf(foo) = 'PString' then
#report error in foo
else
#foo has address
Timer ( [ value [, Boolean ]] )
```

`Timer ( [ value [, Boolean ]])`

Timer uses the global variable TickCount to provide timing-related functions. If you pass no arguments, Timer returns the current TickCount. If you specify a value (typically a previous value of TickCount), Timer returns the difference between that value and the current value of TickCount (that is, `TickCount-value`). If you also specify a nonzero Boolean value, the difference is returned as a string of the form "sss.hh", representing seconds and hundreths of a second. (If the Boolean is zero, it's ignored.)

`TypeOf ( expression )`

TypeOf returns a string containing the name of the type of the given expression. If SADE doesn't know the name of the type, it returns a string of the form "Type #*n* ", where *n* is SADE's internal index for the type.

`Undef` (*parameter* | *variable*)

Undef determines whether the given SADE parameter or variable has been initialized. If the parameter or variable is uninitialized, UnDef returns1; if it's not initialized, Undef returns zero. If you pass an undefined identifier to Undef, an error results.

`Where` (*address*)

Where returns a string containing a symbolic representation of the given address.

Operator precedence

This section describes the operators used to form expressions within SADE. These operators are listed in precedence from highest to lowest. Groupings within the table show operators of the same precedence.

( )		Grouping by parentheses (includes casting, argument lists)
->		Qualifier by pointer (C)
.		Qualifier
++		Increment (C)
--		Decrement (C)
†		Trap
@		Address of (Pascal)
^		Pointer to (Pascal)
¬	NOT !	Logical NOT
~		Bitwise one's complement
*		Pointer to (C)
+		Unary positive
-		Unary negation
&		Address of (C)

*			Multiplication
/	DIV	÷	Division
//	MOD		Remainder
+			Addition
-			Subtraction
>>			Shift right
<<			Shift left
=	= =		Equal
<>	≠	!=	Not equal
<			Less than
>			Greater than
<=	≤		Less than or equal
>=	≥		Greater than or equal
&	AND		Bitwise AND
&&			Logical AND
|	OR		Bitwise OR
| |			Logical OR
XOR	EOR		Bitwise exclusive OR
?:			Condition (C)
:=			Size-compatible assignment
<-			Arbitrary assignment
<op>=			Assignment with operation (C)
..			Range

◆ *Note:* The range operator (..) may only appear once in an expression.

Expression operand base types

SADE provides a number of basic types; you can define additional types as well. The SADE base types are:

Boolean	A 1-byte Pascal Boolean
UnsignedByte, UnsignedChar	A byte in the value range 0 – 255

Byte, Char	A byte in the value range –128 – 127
CChar	A byte in the value range 0 – 255
PChar, PascalChar	A word in the range 0 – 255 (Pascal Char)
UnsignedWord, UnsignedShort	A word in the range 0 – 65,535
Word, Short, Integer	A word in the range –32,768 – 32,767
UnsignedLong, UnsignedInt, Unsigned, UnsignedLongInt	A long word in the range 0 – 4,294,967,295
Long, Int, LongInt, SignedLong, SignedLongInt	A long word in the range –2,147,483,648 – 2,147,483,647
Single, Float, Real	An IEEE floating-point single-precision value (4 bytes)
Double	An IEEE floating-point double-precision value (8 bytes)
Extended	An IEEE floating-point extended-precision value (10 bytes)
Extended12	An IEEE floating-point extended-precision value (12 bytes)
Comp[utational]	A SANE signed 8-byte integer
CString	Up to 254 characters terminated by null byte
PString, String, Str255	A length byte followed by up to 254 characters

▲ **Warning** The SADE basic type Byte is frequently overriden by the Pascal type Byte (which is two bytes long). You can use SizeOf(byte) to tell if this has happened, or you can use the SADE type UnsignedByte instead. ▲

Expression evaluation

This section describes how a single-term or multi-term expression is evaluated by SADE. A single-term expression is represented by a single symbol, and takes on the value represented by the symbol (the value associated with the name, constant, or string). If a symbol represents an array or record structure, the "value" of the expression is the entire array or record structure.

A multi-term expression consists of two or more operands. Multi-term expressions are reduced to a single value according to the following set of rules:

- Each signed integer operand is converted to a 32-bit signed value.

- Each unsigned integer operand is converted to a 32-bit unsigned value.
- Each floating-point and computational value is converted to a 10-byte extended value.
- When a binary operator combines two integer operands, both operands are treated as unsigned if either is unsigned. The result is then treated as a 32-bit unsigned value.
- If both integer operands combined with a binary operator are signed, then the result is a signed 32-bit value.
- If either operand is a floating-point value, then the other operand is converted to floating-point extended and the result is extended.
- Integer division by zero yields zero as the result. Floating-point division by zero yields infinity, except for zero divided by zero, which yields a NaN.
- Operations are performed from left to right, following the precedence indicated earlier in the chapter. Assignment operators are performed from right to left.
- A parenthesized subexpression is reduced to a single value. The resulting value is then used in computing the final value of the expression.
- When parenthesized subexpressions are nested, the innermost subexpression is evaluated first.
- Integer division always yields an integer result; any fractional portion of the result is dropped.
- The logical operators NOT (¬, !), = (==), <> (≠, !=), >, <, <= (≤), >= (≥), &&, | | evaluate to the value 1 (true), and the value 0 (false). Comparison is algebraic, except when character strings are compared.
- The << operator shifts zeros in; bits shifted out are lost. The >> operator maintains the sign of the expression—0 if positive, 1 if negative.
- C programmers should note that pointer arithmetic using the (+), (–), and (<op>=) operators works as in Pascal. Pointer arithmetic using the (++) and (– –) operators works as expected.
- The assignment (:=, <-, <op>=), pointer (^, *), trap (†), and address (@, &) operators are special operators with meanings unique to SADE. They're discussed separately in the sections that follow.

The assignment operator

The assignment operator in SADE is treated as a binary operator. As such it may be embedded in a more complex expression to capture intermediate results. The assignment operator is the only operator that evaluates from right to left. Thus an expression of the form

```
a := b := c := d
```

is evaluated as if it had been written like this:

```
a := (b := (c := d))
```

The left operand of an assignment must be a variable reference. For an integer reference, the right operand is saved in the specified variable. For a floating-point assignment, the right operand is converted to extended before the assignment, if necessary. For string, record, or array assignments, the left variable must be compatible with the right operand, and no other operators may be combined with the assignment.

The above rules also hold for the <op> forms of operators.

Compatibility between operands in SADE is defined as it is in Pascal for real and integer. For structured data, compatible operands are defined as having the same aggregate size. A second operator is provided for *arbitrary* assignment, namely <–. Using this assignment operator, you may assign any type to any other type, regardless of size. For arbitrary assignments, the size of the operand on the right side of the operator is used to determine the number of bytes to move. This operator may be used, for example, to patch memory.

- *Note:* The above restrictions, as well as the discussion of the <– operator, do not apply to SADE variable assignment. SADE variables are dynamically typed, automatically taking on the type of the value assigned to them.

The pointer operator

There are two forms of pointer operators, one following Pascal conventions and another following C conventions. When the Pascal pointer symbol (^) is used as an operator, it follows an expression term, for example `myWindowPtr^`. In the C convention, the pointer symbol (*) precedes the variable reference; for example `*myWindowPtr`.

When the term is a variable reference, the pointer operator indicates an indirect reference through the variable, and the type of the term is determined by the type associated with the pointer variable reference. When the term is a subexpression, the pointer operator indicates an indirect reference through the address represented by that subexpression. The type in this case is assumed to be a pointer to a long integer. Type coercion (described below) may be used to treat the reference as some other type.

The address operator

A pointer to a variable (an address) can be generated with the address operators (@) and (&). Address operators are unary operators taking a variable or procedure reference as the operand. The type of the value is considered as a pointer to the type of the variable. The address operators cannot be applied to SADE variables.

The trap operator

An expression whose value is a trap can be created by using the trap operator (†). The trap operator is a unary operator taking an expression element or a parenthesized expression as the operand. Such trap expressions are used with breakpoint commands to distinguish trap breakpoints from address breakpoints.

Type coercion

Names of known types may be used in a function-like notation to perform type coercions on expressions. The names of types may be predefined base type names or types defined in your program. The type of the object being coerced will be changed as long as there is a reasonable way to interpret and perform the coercion.

Additionally, the type specification may be preceded by the pointer operator (^) to indicate coercion to a pointer to the specified type. This is an extension of the Pascal notation which allows type declarations such as `^integer`. (In fact, you may precede a type name by up to three pointer operators to construct new pointer types.)

The following examples illustrate how the type coercion mechanism works in conjunction with indirect memory references.

Pascal	C	Result
`comp(10)`	`(comp)10`	Converts the number 10 to the comp (computational) type
`comp(10^)`	`(comp)*10`	Converts the long at location 10 to the comp type
`^comp(10)`	`(comp*)10`	Identifies 10 as a pointer to a comp
`^comp(10)^`	`*(comp*)10`	Returns the comp at location 10

Ranges

Certain SADE commands take ranges as arguments. Ranges of addresses or values can be expressed by a pair of expressions (the low and high ends of the range), separated by the range operator (..). The syntax is as follows:

expr .. expr

Neither expression used to designate a range can be a floating-point value. If one end of the range expression is a trap number, both must be; for example, `†$A000..†$AFFF`.

Appendix A **Command Summary**

File commands

Close	Close a file
Open	Open a file
Redirect	Redirect standard output
Save	Save a file

Application control commands

Directory	Display or change the current directory
Kill	Kill an application
Launch	Launch an application
SADEKey	Define a key for entering SADE
SourcePath	Set search path for source files
Target	Select target program for debugging

Menu and Alert commands

Addmenu	Create a menu or add menu items
Alert	Display an alert box
Beep	Specify tones for Alert command
Deletemenu	Delete menus or menu items

Heap commands

Heap	Display heap information
Heap check	Check heap consistency
Heap totals	Display heap summary

Resource commands

Resource	Display the resource map
Resource check	Check the resource map

SADE execution commands

Abort	Stop break action and cancel pending commands
Execute	Execute a script
Quit	Quit SADE
Shutdown	Shut down (with restart option)
Stop	Stop break action execution and execute pending commands

Breakpoint and tracepoint commands

Break	Set breakpoints
List	List processes, breakpoints, tracepoints
Trace	Set tracepoints
Unbreak	Remove breakpoints
Untrace	Remove tracepoints

Program flow control commands

Go	Start execution
Step	Step through code

SADE programming commands

Begin...End	Group commands
Cycle	Continue execution at conditional test of current looping construct
For...End	Loop with a control variable
Func...End	Define a SADE function
If...End	Conditionally execute commands
Leave	Leave current looping construct
Proc...End	Define a SADE procedure
Repeat...Until	Conditionally loop with end test
Return	Return from a SADE procedure or function
While...End	Conditionally loop with beginning test

SADE variable commands

Define	Declare a SADE variable
Undefine	Remove a SADE variable, macro, function, or procedure

Special-purpose display commands

Disasm	Disassemble code
Dump	Display unstructured memory in hexadecimal
Stack	Display stack frames

Miscellaneous commands

Case	Change case sensitivity
Find	Search for a target
Help	Display help information
Macro	Create a macro
Printf	Send formatted output to file or window
Version	Display SADE version number

Appendix B **Program Symbols**

THIS APPENDIX GIVES EXAMPLES OF SCOPING for program symbols. Sample program units, in both C and Pascal, define procedures, functions, and variables. These symbol names are then entered from within various scopes, showing when symbol names need qualification and when they don't. ■

```
/* file MoreStuff.h */
extern int globalint;
extern int globalfunc ();
*/

/* file MoreStuff.c */
#include "MoreStuff.h"
int globalint = 4;
static int localfunc ()
{
      int localint = 3;
      return localint;
}
int globalfunc()
{
      int localint = 3;
      localint = localfunc();
      return localint;
}

/* file Stuff.c */
#include "MoreStuff.h"
static int anotherglobalint = 2;
int main ()
{
      anotherglobalint = globalfunc();
}
/*
```

```
target 'CStuff'
break main.(1)                          # breaks on main
break globalfunc.(1)                    # breaks at globalfunc
break localfunc.(1)
  ### Could not find "localfunc" as a program symbol
break \MoreStuff.localfunc.(1)          # breaks at localfunc

launch 'CStuff'                         # cause stuff to execute

# broke at main.(1)
globalint                               # IF MoreStuff.c built with sym on
  4
globalint   # IF Stuff.c only built with sym on
  ### Could not find "globalint" as a program symbol
anotherglobalint
  2
localint
  ### Could not find "localint" as a program symbol
globalfunc.localint
  ### The variable, "localint", is in a register and cannot be
  ### referenced except in its own frame
localfunc.localint
  ### Could not find "localfunc" as a program symbol
\MoreStuff.localfunc.localint
  ### The variable, "localint", is in a register and cannot be
  ### referenced except in its own frame

# continue executing
go

# broke at globalfunc.(1)
localint
  3
globalint
  4
anotherglobalint
  ### Could not find "anotherglobalint" as a program symbol
\Stuff.anotherglobalint
  2
# continue executing
go

# broke at localfunc.(1)
localint
  4
globalfunc.localint
  ### The variable, "localint", is in a register and cannot be
referenced except in its own frame
```

```
{ file MoreStuff.p }

UNIT MoreStuff;

INTERFACE

VAR
  globalint: LongInt;

  FUNCTION globalfunc: LongInt;

IMPLEMENTATION

  FUNCTION localfunc: LongInt;

    VAR
      localint: LongInt;
      conflicted: Boolean;

    FUNCTION NestedFunction: LongInt;

      VAR
        conflicted: Boolean;

      BEGIN
        conflicted := True;
      END;

    BEGIN
      localint := 5;
      conflicted := False;
      localint := NestedFunction;
      localfunc := localint;
    END;

  FUNCTION globalfunc: LongInt;

    VAR
      localint: LongInt;
    BEGIN
    localint := 3;
    localint := localfunc;
    globalfunc := localint;
    END;

END.
```

```
{ file Stuff.p }

PROGRAM Stuff;

  Uses MoreStuff;

  VAR
    anotherglobalint: LongInt;

BEGIN
anotherglobalint := 2;
globalint := 4;
anotherglobalint := globalfunc;
END.


target 'Stuff'
break Stuff.(3)              # breaks on 3rd statement in Stuff
break globalfunc.(2)         # breaks at globalfunc
break localfunc.(2)
  ### Could not find "localfunc" as a program symbol
break \MoreStuff.localfunc.(2)    # breaks at localfunc
break \MoreStuff.NestedFunction.(2)
  ### Could not find "NestedFunction" as a program symbol
break \MoreStuff.localfunc.NestedFunction.(2)

launch 'Stuff'               # cause stuff to execute

# broke at Stuff.(3)
globalint
  4
anotherglobalint
  2
localint
  ### Could not find "localint" as a program symbol
globalfunc.localint
  ### The variable, "localint", is in a register and cannot be
  ### referenced except in its own frame
localfunc.localint
  ### Could not find "localfunc" as a program symbol
\MoreStuff.localfunc.localint
  ### The variable, "localint", is A6 based and its procedure is not
  ### in the call chain
```

```
# continue executing.
go

# broke at globalfunc.(2)
localint
  3
globalint
  4
anotherglobalint
  2                        # Pascal main globals are really global
\Stuff.anotherglobalint    # in fact, you can't use unit qualifiers
 ### Could not find "anotherglobalint" as a program symbol
\MoreStuff.globalint       # in globals defined in main or unit interface
  ### Could not find "globalint" as a program symbol

# continue executing
go

#broke at localfunc.(2)
localint
  5
globalfunc.localint
  ### The variable, "localint", is in a register and cannot be
  ### referenced except in its own frame
conflicted
  FALSE
# continue executing
go

# broke at NestedFunction.(2)
conflicted
  TRUE
localfunc.conflicted       # because localfunc is a function, references
                           # to localfunc refer to the result of the
                           # function, not its code
  ### No field named "conflicted" in the record

\morestuff.localfunc.conflicted   # so you must use fully qualified name
  FALSE
localint                   # no problem, localfunc.localint is in scope
  5
```

Part II Command Reference

Abort—stop break action and cancel pending commands

Syntax abort

Description The Abort command terminates the current break action and returns you to SADE, canceling all pending commands. This means that if current execution is within a structured statement (Begin...End, for instance), or if multiple commands are selected, these pending commands are not executed. To terminate a break action without canceling pending commands, see the Stop command.

Example

```
#-- Establish target, suspending application
#-- and re-entering SADE at main.(1)

directory 'VolName:some:path:toProg:'
target 'MyProg'
break \MyProg.main.(1)
launch 'MyProg'

#-- Break action examines the current event type
#-- (assumes the event record is named myEvent)
#-- For mouseDown, show co-ordinates of mouseDown and stop.
#-- (Quitting break action but executing rest of pending SADE commmands)
#-- For keyDown, abort. (Quitting break action AND rest of commands)
#-- For any other event, show event type and continue executing
#-- (By default, break actions end with an implicit go)

proc EventFilter
  define global mouseAt;
  if myEvent.what = 1 then     # mouseDown event
    mouseAt := myEvent.where;
    stop;
  elseif myEvent.what = 3 # keyDown event
    "keyDown"; abort;
  else
    printf "Event type %d: \n", myEvent.what;
    printf;
  end
end

Begin
  break _getNextEvent from applzone..applzone^ EventFilter
  go
  printf "Mouse down at H: %d\n          V: %d\n", mouseAt.h, mouseAt.v;
end
```

```
#-- output
Event type 0:
Event type 8:
Event type 6:
Mouse down at H: 208
               V: 144
#-- with same break action in place, restart target.
go
#-- output
Event type 6:
keyDown
```

See also Break, Stop

AddMenu—create a menu or add menu items

Syntax addmenu [*menuname* [*itemname* [*command*]]]

Description The AddMenu command lets you create menus and add menu items to execute SADE commands. *Menuname*, *itemname*, and *command* are all string expressions and must be enclosed in quotation marks if they are string constants.

If a menu of *menuname* does not exist, a new menu is created. If a menu item with the specified *itemname* already exists, it's replaced; otherwise a new menu item is created. You can include a Command-key equivalent for the item by listing the command key after a slash (/) at the end of the string.

If *itemname* or *command* are not specified, AddMenu returns the current value from the specified level down. For instance, if *itemname* is specified without any commands, AddMenu displays the command that's currently defined for that menu item. If *menuname* is omitted as well, SADE returns the current values for all user-defined menus and menu items.

Examples

```
addmenu 'Debug' 'Disasm/1' 'disasm'
addmenu 'Debug' 'Code Resources' 'heap restype "CODE"'
```

See also DeleteMenu

Alert—display an alert box

Syntax alert [beep] *message*

Description The Alert command displays an alert box containing the specified message. *Message* is a string expression and must be enclosed in quotes if it's a string constant. The alert is displayed until the OK button is clicked. If **beep** is specified, a sound is generated when the alert box appears.

Example

```
if length(str) > 64 then
  alert "string longer than expected"
end
```

See also Beep

Beep—generate tones

Syntax

beep [*notespecs*]

where

notespecs is a string of the form

[*note*[,*duration*[,*level*]] ...]

Description

For each *notespec*, the Beep command produces the given note for the specified duration and sound level. Multiple *notespecs* are separated by blanks or tabs. If no *notespecs* are given, a simple beep is produced.

Note is one of the following:

- A number indicating the count field for the square wave generator, as described in the Summary of the Sound Driver chapter of *Inside Macintosh*.
- A string in the following format:

  [*n*] *letter*[# | b] where *n* is an optional number indicating the octaves below or above middle C, followed by a letter indicating the note (A–G) and an optional sharp (#) or flat (b) character.

The optional *duration* is given in sixtieths of a second. The defualt duration is 15 (one-quarter second).

The optional sound *level* is given as a number from 0 to 255.
The default level is 128.

Example

```
beep "2C,20 '2C#,40' 2D,60"
#-- Play the 3 notes specified: C, C sharp, and D, all two octaves above
#-- middle C, for one-third, two-thirds, and one full second,
#-- respectively. Note that the second parameter must be quoted;
#-- otherwise,the sharp character would indicate a comment.
```

See also Alert

Begin...End—group commands

Syntax

begin
commands
end

Description

The Begin...End construct allows a sequence of commands to be grouped together or bracketed. One use of this construct is to specify a breakpoint action consisting of multiple commands or procedure calls.

Example

```
break DisplayString.(4) begin
  str := theStr^   # save value of parameter in variable str
  if str = '***' then
    stop
  end
end
```

Break—set breakpoints

Syntax

break *addr*,... [*break-action*]
or
break *trap* [from *addr-range*],... [*break-action*]
or
break *trap-range* [from *addr-range*],... [*break-action*]
or
break all traps [from *addr-range*] [*break-action*]

Description

The Break command sets one or more breakpoints within a target program's code. There are two types of breakpoints: address breakpoints and trap breakpoints. Both kinds of breakpoints may be followed by a break action, a command (or series of commands) that's executed when the breakpoint is reached.

You can set an address breakpoint anywhere within your program by specifying a RAM address. You can also use symbolic references, in which case the code need not be in memory at the time the breakpoint is set.

Trap breakpoints can be set on a single trap, a range of traps, or on all traps. Traps can be specified by either trap name or trap number. Trap numbers must be prefixed with the trap character (†) and trap names must be preceded by an underscore (_InitGraf, for example). You can also specify a memory range, in which case SADE breaks only when the trap is called from the specified range.

The same trap may be specified in multiple break commands; you can use the List command to see what breakpoints have been set. When implementing trap breakpoints, SADE first looks for a call that matches a specified trap name, checking the address range if one was given. If no match is found, SADE then looks for trap ranges containing the trap. SADE takes the most recently defined range containing the trap, again checking the address range if one was specified.

- *Note:* If you set an address break on an instruction that is a trap call for which a trap break has already been set, the address break is recognized and the trap break is not.

If a break action is specified, SADE resumes program execution after executing the command(s). There are two ways to suspend program execution as part of a break action. The Abort command returns to SADE immediately, canceling any pending commands. The Stop command executes any pending commands and then returns to SADE.

▲ **Warning** The commands specified in the breakpoint action are saved and are not interpreted until the breakpoint is reached. You should be sure that program symbol references will be correctly interpreted at the time the breakpoint is reached. In other words, a reference may be fine at the time you define a break action, but be out of scope when the action is actually executed.▲

◆ *Note:* If no break action is specified *and* the execution command (Go, for instance) just prior to hitting a breakpoint is part of a structured statement (such as While...End), or if multiple commands were selected, the remaining commands are executed after hitting the breakpoint and re-entering SADE.

You can specify multiple breakpoints, separated by commas, with a single Break command. If this command includes a break action at the end, the action is applied to all breaks on the list.

Examples

```
break _GetResource
break †$A997..†$A9A0
break all traps from myproc.(1)..myproc.(5)
break all traps from applZone..applZone^

#-- The following example sets multiple breakpoints.

Break procB.(1), _LineTo from procA.(1)..procA.(1000), _setPort Begin;"hit one"; end

list break
  procB.(1)    # $586398  # processID =5  # has break action
  _LineTo  # processID =5  # †$A891  # called from procA.(1) .. procA.(83)  # has break action
  _SetPort  # processID =5  # †$A873  # has break action

#-- A break set on top of a matching breakpoint replaces the older one.
#-- Breakpoints match if they are set on the same address or trap and
#-- the called from address range, if there is one, matches.

Break procB.(1)
break _lineTo
list break
  procB.(1)    # $586398  # processID =5
```

```
_LineTo  # processID =5  # †$A891
_LineTo  # processID =5  # †$A891  # called from procA.(1) .. procA.(83)  # has break action
_SetPort # processID =5  # †$A873  # has break action
```

See also Abort, List, Stop, Trace, Unbreak

Case—change case sensitivity

Syntax case on | off

Description By default, case sensitivity is **on**, which means that when looking up symbols, SADE first performs a case-sensitive lookup. If the symbol isn't found, SADE converts all the characters to uppercase and looks again.

C programmers will want case sensitivity turned on. Pascal programmers, however, may want to set case sensitivity **off**; SADE then converts all symbols to uppercase before any lookup, speeding up the search process.

Examples

```
#-- C programmers
case on
CFunction
CFunction.(0)
CFUNCTION
### Could not find "CFUNCTION" as a program symbol
case off
CFunction
### Could not find "CFunction" as a program symbol
#-- Pascal programmers
case on
PascalProc
PASCALPROC.(0)
PASCALPROC
PASCALPROC.(0)
case off
PascalProc
PASCALPROC.(0)
```

Close—close a file

Syntax close [all | *windowName*]

Description The Close command closes the specified file or all files. *WindowName* is a string expression and must be enclosed in quotation marks if it's a string constant.

If no parameters are given, Close closes the target window. Note, however, that the SADE Worksheet file cannot be closed. If the contents of a file have not been saved, a dialog box asks whether they should be.

Examples

```
close 'myFile'
close targetWindow   #no quotes needed--uses SADE variable TargetWindow
```

Cycle—continue execution within construct

Syntax cycle [if *Boolean*]

Description The Cycle command causes execution to continue from the conditional test of a While, Repeat, or For construct, or from the beginning of a Loop construct. If a Boolean expression is specified, Cycle executes only if the expression is nonzero; otherwise execution continues immediately following the Cycle command.

Example

```
define testnum := 0
define endnum := 6
define cycleMax := 4

repeat
  printf "testNum = %d\n", testNum
  testNum := testNum + 1
  cycle if testNum < cycleMax
  "\nDidn't cycle"
until testNum = endNum

#-- output

testNum = 0
testNum = 1
testNum = 2
testNum = 3

Didn't cycle
testNum = 4

Didn't cycle
testNum = 5

Didn't cycle
```

See also Leave

Define—declare a SADE variable

Syntax

define [global] *declaration* [,...]

where *declaration* has the form

name [*dimension*] [:= *init-value* | = *init-value*]

where

name must be unique in the current scope unless declared global.

dimension is an *expr* enclosed in brackets [ ]

init-value is either an *expr* for the initial value of simple types, or a list of the following form for structured types:

([*expr* of] *init-value* , ..)

where the optional **of** clause allows for replication of a value.

Description

The Define command defines one or more SADE variables. A variable must be defined before it's used. The variable declaration identifies the name, scope, and (optionally) the initial value of the variable. Multiple declarations must be separated by commas.

SADE variables are dynamically typed; that is, their type is determined on assignment (and may be changed by new assignments). SADE variables defined as arrays require an index. SADE array variables may contain a heterogeneous set of values; that is, the elements may contain values of different types.

An initial value for simple types may optionally be specified by an *expr* following the assignment operator (:=) or, in this case only, (=). If the declared item is an array, a list of initial values may be specified as the values of the array elements.

The scope of a variable can be either global or local. If a variable is defined outside a procedure (or function), its scope is automatically global. In other words, it is known both inside and outside of any procedures. If a variable is declared inside a procedure, its scope is local unless the keyword **global** is given. If a global and a local variable exist with the same name, the local symbol overrides the global symbol.

Redefining global variables replaces the previous definition, with one exception: If the definition is within a procedure, and the new definition matches the existing definition, then the existing definition is retained. For example, when a global variable is defined within a procedure or function and is given an initial value, the initialization occurs only when the variable is actually created. Subsequent invocations of the procedure do not affect the current value of the global variable, and can make use of the value left in the variable by the preceding invocation.

The Define command may not be used within any structured statement. To remove a variable definition, use the Undefine command.

Examples

```
#-- define a five element array, with the first four elements true
#-- and the last element false

define global test[5] := (4 of 1,0)

#-- define a 30 element array, with the first 29 elements true
#-- and the last element false

define arraysize := 30
define myArray[arraysize] := (arraysize-1 of 1,0]

#-- In the next example, note that the definition of INDEX in
#-- RotateLeftOneWCarry() is local to that procedure; otherwise it
#-- would be attempting to redefine the global INDEX that's used
#-- as a FOR loop counter when RotateLeftOneWCarry() is called.

proc RotateLeftOneWCarry()
  define global SADEArray[4] := ("indexed by one", 2, 3.33, "fourth and last")
  define index
  define holder
  for index := 1 to 4 do
    if index = 1 then
      holder := SADEArray[index]
    else
      SADEArray[index-1] := SADEArray[index]
      if index = 4 then
        SADEArray[index] := holder
      end
    end
  end
end

define index
define holder
for index := 1 to 4 do
  RotateLeftOneWCarry()
  for holder := 1 to 4 do
    SADEArray[holder]
  end
  printf "\n"
end
undefine RotateLeftOneWCarry
```

```
#-- output

2
3.33
fourth and last
indexed by one

3.33
fourth and last
indexed by one
2

fourth and last
indexed by one
2
3.33

indexed by one
2
3.33
fourth and last
```

See also Undefine

DeleteMenu—delete menus or menu items

Syntax deletemenu [*menuname* [*itemname*]]

Description The DeleteMenu command deletes menus and/or menu items. *Menuname* and *itemname* are string expressions and must be enclosed in quotation marks if they are string constants. If only *menuname* is specified, the entire menu is deleted. If a user-defined menu item with the specified name exists, it is deleted. The standard SADE menus and menu items cannot be deleted.

▲ **Warning** If both *menuname* and *itemname* are omitted, all user-defined items are deleted. ▲

Example

```
deletemenu "Special" "Launchapp"
```

See also AddMenu

Directory—set or write the default directory

Syntax directory [*directoryname*]

Description When you first enter SADE, the default directory is the directory where SADE resides. The Directory command sets the default directory for all SADE file-oriented operations to the specified directory. *Directoryname* is a string expression and must be enclosed in quotation marks if it's a string constant. If *directoryname* isn't specified, the current default directory is displayed.

Examples

```
macro here "volume:very:long:directory:path"
directory here
```

```
macro RootPath "volume:very:"
directory concat(RootPath, "long")
```

See also Sourcepath

Disasm—disassemble code

Syntax

disasm [*addr* [*count*]]
or
disasm [*addr-range*]

Description

The Disasm command disassembles instructions starting at the location specified by *addr* or *addr-range*. The default behavior when no address is specified is to begin disassembling at the end of the last disassembly. If the value of the program counter has changed since the last disassembly, the program counter (PC) is used as the default starting address. If no range or count is specified, the number of instructions (not lines) disassembled defaults to 20.

Each line of the disassembly output is divided into four fields: the module offset, the address of the instruction, the hexadecimal encoding for the instruction, and the assembly code (opcode, operand, and comment). You can modify the presence, order, and format of these fields by changing the value of the built-in variable DisAsmFormat (described in Chapter 3).

Example

```
#-- diassemble 5 instructions in standard format, starting at
#-- the eighth statementof the DisplayText routine.

disasm DisplayText.(8) 5

#-- output

DisplayText
+0040   003191A8  2F2D FE64    MOVE.L  -$019C(A5),-(A7)
+0044   003191AC  2F2D FE78    MOVE.L  -$0188(A5),-(A7)
+0048   003191B0  4EBA FE26    JSR     FlushDWindow    ; 00318FD8
+004C   003191B4  486D FE48    PEA     -$01B8(A5)
+0050   003191B8  4EBA 00C0    JSR     DisplayString   ; 0031927A
```

See also Dump

Dump—display memory

Syntax

dump [byte | word | long] [*addr* [*count*]]
or
dump [byte | word | long] [*addr-range*]

Description

The Dump command displays memory at the location specified by *addr* or *addr-range*. If no parameter is given, the memory starting at the program counter is displayed. The memory is displayed in hexadecimal and ASCII characters according to the specified format, which may be byte, word, or long. The default format is word.

Remember that to dump the value of a variable "foo", you must specify `dump @foo` (since `dump foo` would take foo's value and use it as an address).

Examples

```
dump byte a5
$00146A8E 00 14 68 74 FF FF FF FF  00 00 00 00 00 00 00 00  ..ht............
dump word a5 40

$00146A8E 0014 6874 FFFF FFFF  0000 0000 0000 0000  ..ht............
$00146A9E 000E 6C66 0000 FFFF  FFFF FFFF FFFF FFFF  ..lf............
$00146AAE 0001 4EF9 000E 6E00                       ..N...n.

dump long a5..a5+40

$00146A8E 00146874 FFFFFFFF  00000000 00000000  ..ht............
$00146A9E 000E6C66 0000FFFF  FFFFFFFF FFFFFFFF  ..lf............
$00146AAE 00014EF9 000E6E00  00                 ..N...n..
```

See also

Disasm

Execute—execute commands in a file

Syntax execute *filename*

Description The Execute command executes the commands and definitions contained in the specified file. *Filename* is a string expression and must be enclosed in quotation marks if it's a string constant. The Execute command can't be used within a structured statement.

Example

```
#-- In this example, the Redirect command creates a file to hold
#-- output from SADE. Entering a string echos the string. That
#-- output is redirected to the file, becoming the file's contents.
#-- Here the string is a comment and a SADE command to execute
#-- the contents of the next file in the chain.

open 'exec1'
redirect 'exec1'

  '"\n executing exec1 now"'
  "execute 'exec2'"

  open "exec2"
  redirect 'exec2'
    "'now executing exec2'"
    "execute 'exec3'"

    open "exec3"
    redirect 'exec3'
      '"Done in exec3"'

redirect pop all
execute "exec1"

Alert "Try a tile windows here\n∂
   Then look at the worksheet for output"

#-- output

executing exec1 now
now executing exec2
Done in exec3
```

Find—search for a target

Syntax

find [count] *target* [,*n*] [*addr-range* [mask *mask*]]
or
find [count] *target* [,*n*] [*addr* [*count*] [mask *mask*]]

Description

The Find command searches memory for a target pattern, which can be either a numeric or string expression. If you specify the **count** keyword, Find tells you how many occurences of the target it found. If you omit the **count** keyword, Find displays the address of the first occurrence of the pattern. You can use the *n* parameter to specify which occurrence of the pattern you want; for instance, if you specify 3 for *n*, Find returns the address of the third occurrence of the pattern.

You may start the search at a specified *addr* and look up to *count* bytes beyond, or you may limit the search to *addr-range*. The default range is the MultiFinder block containing the application's heap and stack (in other words, all of the memory that belongs to the application).

The *mask* parameter (prefaced by the **mask** keyword) is an optional numeric or string expression that is logically ANDed with the contents of each memory location before the comparison is done.

△ **Important** Remember that expression values in SADE are long words by default; to specify another size, use typecasting, as shown below. △

Examples

```
dump $20 $40
$00000020 0027 A002 0027 A00A  0040 1F52 0027 A012  .'...'...@.R.'..
$00000030 0027 A01A 0027 A032  0040 113C 0027 A02A  .'...'.2.@.<.'.*
$00000040 0040 113C 0040 113C  0040 113C 0040 113C  .@.<.@.<.@.<.@.<
$00000050 0040 113C 0040 113C  0040 113C 0040 113C  .@.<.@.<.@.<.@.<

#-- Looking for the number of occurances of target,
#-- from start address, for number of bytes.
#-- Target is cast to word size. The default size would
#-- be long and nothing in this range matches $0000113C.

find COUNT (word)$113C $20 $40
9
```

```
#-- Looking for first occurance of target, in address range.
#-- C style type cast to limit the target to word size.

find (word)$113C $20..$96
$0000003A

#-- Looking for second occurance of target.
#-- Using Pascal style typecast on target.

find word($113C), 2 $20 $40
$00000042

#-- Looking for a long.
#-- Masking the memory searched prior to the compare.

find $0000113C $20..$96 MASK $0000ffff
$00000038

#-- Match same last target with same last mask.
#-- Just change the address range to search.

find SAME $40..$50
$00000040
```

For...End—loop with a control variable

Syntax

for *clause* [do]
  commands
end

where *clause* may have one of the following forms:

var := *expr* to *expr*
var := *expr* downto *expr*
var := *expr* , ...

Description

The For...End construct provides looping with a control variable. The enclosed commands are executed until the control variable has taken on each successive value in the range expressed by *clause*.

The control variable *var* must be declared before it can be used, and array variables are not allowed. For the clause *expr* **to** *expr*, the commands are executed and the control value incremented once for every integer value in the range. For the clause *expr* **downto** *expr*, the control value is decremented. The third clause is a list of expressions; execution continues until the control variable has taken the value of each of the listed expressions.

For...End constructs may be nested; they may also be used within other flow-control constucts, as well as in break actions. The control variable may be modified within the body of the loop (but cannot, of course, be shared between nested constructs).

Example

```
define var := 0
define outerLooper, syncopation
for outerLooper := 3 downto 1 do
  "\n"
  for syncopation := "one","two","three", var
    printf "%d -- " , outerLooper
    syncopation
    var := var + 1
  end
end
#-- output
3 -- one
3 -- two
3 -- three
3 -- 0
2 -- one
2 -- two
2 -- three
2 -- 4
1 -- one
1 -- two
1 -- three
1 -- 8
```

Func...End—define a SADE function

Syntax

func *name* [(*arg-name*,...)]
  commands
end

Description

SADE functions are delimited by the Func...End construct. The last statement to be executed must be a Return command specifying a return value. The type of a function is not specified in the definition but rather takes on the type of the value returned. (Thus functions are not limited to returning results of a single type.)

SADE functions use conventional calling notation, with the function name followed by a list of parameters enclosed by parentheses. Function parameters are handled in the same fashion as procedure parameters, and the predefined SADE variables Arg and NArgs may be used. (See the description of the Proc command for an example using these variables.) User-defined functions may be called anywhere an expression is allowed.

Example

```
func fact(n)
  if n <= 1.0 then
    return 1.0
  else
    return n * fact(n-1)
  end
end
```

See also

Proc, Return

Go—resume execution

Syntax

go [til *addr*,...]
or
go [while *expr*]
or
go [until *expr*]

Description

The Go command resumes program execution at the current program counter. If you specify the keyword **til**, SADE sets a temporary breakpoint at the specified address(es). When the breakpoint is encountered, SADE is reentered and the breakpoint is removed. Note that if multiple breakpoints are specified with the keyword **til**, all breakpoints are removed when any of them is reached.

If the address is in ROM, SADE informs you that it can't set a breakpoint in ROM. If a conditional expression is specified, SADE uses trace mode until the condition is met (for the keyword **until**) or broken (for the keyword **while**).

Examples

```
go til \OtherCompilationUnit.myProc.(1)
go while particularVar < 2
go until ProgVar[1] = 3
```

See also Stop

Heap—display heap information

Syntax heap [display] [*addr*] [*blocktype*]

Description The Heap command displays information about the specified heap. You can specify a heap that starts at *addr;* if no address is specified, the heap pointed to by the global variable theZone is displayed. By default, this information is displayed:

- a dot if the object is locked or nonrelocatable
- the block length
- the block type (relocatable, nonrelocatable, free)
- the address of the beginning of the block
- block attributes (locked, resource, purgeable)
- the address of the master pointer if it's a relocatable block
- for standard toolbox data structures, a description of the structure
- for a resource, the resource type and ID, and the reference number of the file it's in

You can specify one of the following *blocktypes* to limit the display to a particular type of block:

- **purge[able]** limits the display to purgeable blocks.
- **nonreloc[atable]** limits the display to nonrelocatable blocks.
- **reloc[atable]** limits the display to relocatable blocks.
- **free** limits the display to free blocks.
- **lock[ed]** limits the display to locked blocks.
- **res[ource]** limits the display to resources.
- **restype** *type* limits the display to the specified resource *type.*

Note that *type* is case-sensitive and should be enclosed in single quotation marks ('MENU', for example).

Example

```
Heap restype 'MENU'
#-- output
BlkAddr    BlkLength Typ MasterPtr  Flags RType   RId RFRef  RName
$00316590  $00000098  H  $0031452C    R   MENU   1000 $0584  "File"
$00316838  $00000050  H  $00314528    R   MENU   1001 $0584  "Edit"
$00316888  $000000F4  H  $00314524    R   MENU   1002 $0584  "Log"
```

See also Heap Check

Heap Check—check heap consistency

Syntax heap check [*addr*]

Description The Heap Check command checks the consistency of the current application heap, which is by default the heap referenced by the global variable theZone. You can specify another heap, but *addr* must be the address of the heap zone header. (In other words, you can't check part of a heap.)

Heap Check performs range checking to make sure all pointers are even and non-NIL, and that block sizes are within the range of the heap. It verifies that the self-relative handle points to a master pointer referring to the same block. For nonrelocatable blocks, it checks if the heap zone pointer points to the zone where the block exists. Heap Check also verifies that the total amount of free space is equal to the amount specified in the header, and that all pointers in the free master pointer list are in the heap.

Examples

```
Printf "Checking %P's heap at $%.08X\n", ^Pstring( $910)^, TheZone
Heap Check TheZone
Printf "Checking the System heap at $%.08x\n", SysZone
Heap Check SysZone
Printf "Checking the Multifinder heap at $%.8x\n", **$2a6+$c
Heap Check $2a6^^+$c
#-- output
Checking SADE's heap at $00146EF2
  The heap is okay.
Checking the System heap at $00001400
  The heap is okay.
Checking the Multifinder heap at $00023d64
  The heap is okay.
```

See also Heap, Heap Totals

Heap Totals—display heap summary

Syntax heap totals [*addr*] [*blocktype*]

Description The Heap Totals command summarizes the state of the current application heap, which is by default the heap referenced by the global variable theZone. You can specify another heap that starts at *addr*.

Information is given for free, nonrelocatable, and relocatable objects. If you wish to restrict the display to a particular type of block, you can specify one of the following *blocktypes:*

- **purge[able]** limits the display to purgeable blocks.
- **nonreloc[atable]** limits the display to nonrelocatable blocks.
- **reloc[atable]** limits the display to relocatable blocks.
- **free** limits the display to free blocks.
- **lock[ed]** limits the display to locked blocks.
- **res[ource]** limits the display to resources.
- **restype** *type* limits the display to the specified resource *type*.

Note that *type* is case-sensitive and should be enclosed in single quotation marks ('CODE', for example).

Example

```
heap totals
#-- output
                              Total Blks  Total Size
Free                                23       49080
Nonrelocatable                       7        1348
Relocatable                         89       21232
  Locked & NonPurgeable              2        5796
  Locked & Purgeable                 2        8136
  UnLocked & Purgeable               6         680
  UnLocked & NonPurgeable           79        6620
Heap (total)                       119       71660
```

See also Heap, Heap Check

Help—display help information

Syntax help [*identifier*,...]

Description The Help command displays information about using SADE, including the syntax of all SADE commands. To see what values *identifier* can have, just enter `help`.

If...End—conditional execution of commands

Syntax

if *Boolean* [then]
  commands
[elseif *Boolean* [then]
  commands] ...
[else
  commands]
end

Description

The If...End construct allows for conditional execution of sequences of SADE commands. Each **if** must be concluded by a corresponding **end**. **Elseif** and **else** are optional, but must appear between the **if** and **end** in the order indicated above. More than one **elseif** may appear, but at most one **else** may appear.

The commands controlled by an **if** extend to the corresponding **end**, or to the first corresponding **elseif** or **else**. The commands controlled by an **elseif** extend to the next corresponding **elseif**, **else**, or **end**. The commands controlled by an **else** extend to the corresponding **end**.

When an If...End construct is evaluated, if the **if** *Boolean* is true, the statements controlled by the **if** are executed and the remainder of the construct to the **end** is skipped. If the *Boolean* is false, the statements controlled by the **if** are skipped and the next (**elseif**) condition is checked, if present. If an **elseif** condition is evaluated and is true, the commands it controls are executed and the remainder of the construct is skipped. If no conditions are evaluated as true, when the **else** command is reached (if present), the commands controlled by the **else** are executed (otherwise, they're skipped).

If...End constructs may be nested.

Example

```
#-- Steps through a five element array using a low to high index,
#-- looking for first true element, and resetting it to false.
#-- Done 5 times, starting with the first four elements true
#-- and the last element false.
define global test[5] := (4 of 1,0)

proc IfDemo
  define whichtest
  define showMe
  for whichtest := 1 TO 5 do
    printf "\n"
    for showMe := 1 to 5 do
      printf "%t", test[showMe]
    end
    printf " - "
    if test[1]
      "test[1] true"
    elseif test[2]
     "test[2] true"
    elseif test[3]
     "test[3] true"
    elseif test[4]
     "test[4] true"
    elseif test[5]
     "test[5] true"
    else
     "all tests false"
    end
    test[whichtest] := 0
  end
end
IfDemo
#-- output
11110 - test[1] true
01110 - test[2] true
00110 - test[3] true
00010 - test[4] true
00000 - all tests false
```

Kill—kill an application or tool

Syntax kill *filename*

Description If you want to terminate an application, (whether it's suspended or not), use the Kill command. Be aware, though, that Kill is dangerous, since it doesn't give the unlucky application a chance to perform its usual exit routines (like saving data). Kill does perform cleanup activities like freeing the MultiFinder memory occupied by the application and removing its trap patches.

Filename is a sting expression and must be enclosed in quotation marks if it's a string constant.

Examples

```
kill 'VolName:full:path:to:targetProg'

define RootPath := "VolName:full:"
kill Concat(RootPath, "path:to:targetProg")

directory 'VolName:full:path:to:'
kill "targetProg"
```

Launch—launch an application

Syntax launch *filename*

Description The Launch command launches the specified application or tool. *Filename* is a string expression and must be enclosed in quotation marks if it's a string constant. Launch does nothing if the file type of the specified file is not 'APPL'.

Examples

```
#-- define directory location for target program sources
#-- (if not in the same directory as the program file)

sourcepath 'VolName:path:targetProg:sources:'

#-- define directory location of target program executable file
#-- and, if different, the location of its .sym file

target 'VolName:path:targetProg' using 'VolName:path:targetProg.sym'

#--- then launch the target progam, as in the examples below,
#--- SADEKey to suspend and begin debugging it.

launch 'VolName:path:targetProg'

#-- or

define RootPath := "VolName:"
launch Concat(RootPath, "path:targetProg")

#-- or

directory 'VolName:path:'
launch "targetProg"
```

Leave—exit from a looping construct

Syntax leave [if *Boolean*]

Description The Leave command lets you exit from a Loop, While, Repeat, or For construct. You can specify a condition for leaving with the **if** keyword.

Example

```
define testnum := 0
repeat
  Printf "testNum = %d\n", testNum
  leave if testNum = 3
  testNum := testNum + 1
until testNum = 6
Printf "after leaving Repeat loop - testNum = %d\n", testNum

#-- output

testNum = 0
testNum = 1
testNum = 2
testNum = 3
after leaving Repeat loop - testNum = 3
```

See also Cycle, For, Loop, Repeat, While

List—list processes, tracepoints, and breakpoints

Syntax

list process
or
list trace [traps | addrs]
or
list break [traps | addrs]

Description

The List command lets you see what processes, breakpoints, and tracepoints are around.

For processes, the display includes the following information: a process number, a "loaded" or "not-loaded" designation, and the filename and symbol filename (typically the filename followed by .SYM) for the process.

For breakpoints and tracepoints, List displays the location along with its symbolic representation. The optional **traps** or **addrs** keywords limit the display to trap or address breakpoints (or tracepoints) respectively.

Examples

```
list break          # list all breakpoints set
#-- output
  EVENTLOOP.(5)     # $21A4E2  # processID =10  # has break action
  EVENTLOOP.(11)    # $21A500  # processID =10
  EVENTLOOP.(2)     # $21A4C4  # processID =10
list process
#-- output
   Process#  Loaded?      FileName, SymbolFile
          6  Loaded       SADE,  SADE.SYM
          2  Loaded       Finder, Finder.SYM
```

See also Trace, Break

Loop...End—repeat commands until Leave

Syntax

loop
commands
end

Description

The Loop...End construct provides unconditional looping. The enclosed commands are executed repeatedly. To exit the loop, use the Leave command.

Loop constructs may be nested.

Example

```
define Inner := 0
define Outer := 0
loop
  loop
    Inner := Inner + 1
    printf "Inner: %d " , Inner
    leave if Inner > Outer
  end
  Inner := 0
  Outer := Outer + 1
  printf "Outer: %d\n", Outer
  leave if Outer > 4
end
#-- output
Inner: 1 Outer: 1
Inner: 1 Inner: 2 Outer: 2
Inner: 1 Inner: 2 Inner: 3 Outer: 3
Inner: 1 Inner: 2 Inner: 3 Inner: 4 Outer: 4
Inner: 1 Inner: 2 Inner: 3 Inner: 4 Inner: 5 Outer: 5
```

See also Leave

Macro—define a macro

Syntax macro *name string-expr*

Description The Macro command associates a string of characters with a name. Macros let you define short, familiar names to use instead of long, unfamiliar strings. For instance, the SADEStartup file defines macros that let you use MacsBug-like syntax for certain SADE commands.

Macro definitions can be nested; that is, they can contain references to other macros. Macro definitions cannot be recursive, however; in other words, a macro definition can't reference itself. Macro definitions are not allowed in structured statements. Macros may be redefined. Macro definitions are limited to a length of 254 characters.

Examples

```
macro br 'break'
macro clr  'unbreak all'
macro dir "volume:very:long:directory:path"
directory dir
```

OnEntry—set commands for SADE entry

Syntax onEntry [*break-action*]

Description The OnEntry command supplies commands that are to be executed each time SADE is entered. Each OnEntry command replaces the commands specified by the previous OnEntry command. In the SADEStartup file, the `StandardEntry` procedure is specified as an OnEntry break action. You can define your own OnEntry actions in the SADEUserStartup file, but you'll probably want to use the one in SADEStartup as a model (so as not to lose the operations it performs).

The OnEntry command accepts only one command or procedure invocation. If you want to execute multiple commands and/or procedures upon entry, use the Begin..End construct to group them.

See also Break, Begin

Open—open a file

Syntax open [source] [behind] *filename*

Description The Open command opens the specified file. *Filename* is a string expression and must be enclosed in quotation marks if it's a string constant. The file must be of type 'TEXT'. The **source** keyword opens the window read-only. If you specify **behind**, the window is opened just behind the frontmost SADE window, otherwise, it's opened as the frontmost window.

Example

```
open 'myFile'
```

See also Close, Save

Printf—print formatted output

Syntax

printf [*format* [, *argument*] ...]
or
printf [(*format* [, *argument*] ...)]

Description

The Printf command is the most complicated of the SADE commands. If you find its myriad options intimidating, remember that Printf has one simple and important use: printing anything you want to a SADE window or file. You can also use Printf to convert and format values in just about any way you might want.

Format is a string containing characters to print, as well as format specifications for arguments that follow. Format specifications are preceded by the % character. For instance, to print the value of the variable myVar as a decimal number, along with a message, enter:

```
printf "The value of myVar is %d" myVar
```

Each format specification applies to zero or more arguments. When the format specifications are exhausted, any remaining arguments are ignored. Likewise, when the specified arguments are exhausted, any remaining format specifications are ignored.

The following fields are used in the format specification:

% [flags] [width] [precision] op

flags — An optional sequence of characters which modify the meaning of the main (op) conversion specification:

- `-` — Left-justify within the field width rather than right-justify if the converted value has fewer characters than the specified minimum field width.
- `+` — Always generate a "+" or "-" sign when converting signed arguments. Note, that negative values are always preceded by a "-" regardless of whether the "+" flag is specified.
- space — Generate a space for positive values and "-" for negative values. This space is independent of any padding used to left or right-justify the value. The "+" flag has precedence over the space flag.
- `#` — Modify the main conversion operation. The modifications performed are described in conjunction with the relevant main conversion operations discussed later.

width — An optional ***minimum*** field width, specified as a decimal integer constant (that doesn't begin with a "0") or an "*". In the latter case a corresponding argument specifies the minimum field width. If the converted value has fewer characters than the width, it is padded to the width on the left (default) or right (if the "-" flag is specified) with spaces (default). If the converted value has more characters than the width, the width is increased to accommodate it. For %t conversions, the width specifies the minimum width to reserve for RECORD type field names.

precision — The optional precision is specified as a "." followed by an *optional* decimal integer or as an "*". In the latter case a corresponding argument specifies the repetition count. If the decimal integer or "*" following the "." is omitted, the precision is assumed to be 0. Precision is used to control the number of digits to be output for numeric conversions or characters for string conversions. Omitting the precision has a default value which is a function of the main conversion to be performed.

op — The required main conversion operation specified as one of the following single characters:

d — The corresponding parameter is converted to a ***signed*** decimal value (floating point values are truncated).

precision — The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags

-	left-justify
+	explicit "+" or "-"
space	space for positive value
#	*ignored*

u — The corresponding parameter is converted to an ***unsigned*** decimal value (floating point values are truncated).

precision — The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	*ignored*

X The corresponding argument is converted to an *unsigned hexadecimal* value. The number of bytes converted is a function of the arg's type. The letters `abcdef` are used for x conversion and `ABCDEF` are used for X conversion.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	prefix converted value with a "$"

b The corresponding argument is converted to an *unsigned binary* value. The number of bytes converted is a function of the arg's type.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	*ignored*

o The corresponding argument is converted to an *unsigned octal* value. The number of bytes converted is a function of the argument's type.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	prefix converted value with a "0"

f The corresponding argument is converted to a signed decimal *floating point* value. The value is converted to the form "[-]ddd.ddd", "[-]INF", or "[-]NAN(ddd)" (where ddd is the NAN code) depending on the value.

precision The precision specifies the number of digits after the decimal point. If the precision is 0, no decimal point appears (which can be overridden with the "#" flag). The default precision is 6.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it

E The corresponding argument is converted to a signed decimal *floating point* value. The value is converted to the form "[-]d.ddde±dd" (for e conversion), "[-]d.dddE±dd" (for E conversion), "[-]INF", or "[-]NAN(ddd)" (where ddd is the NAN code) depending on the value. The exponent always contains at least two digits.

precision The precision specifies the number of digits after the decimal point. If the precision is 0, no decimal point appears (which can be overridden with the "#" flag). The default precision is 6.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it

G The corresponding argument is converted to a signed decimal *floating point* value. The value is converted using f *or* e conversion (or in the style f or E conversion when G is specified). The form of conversion depends on the value being converted; e or E conversion is performed only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result (which can be overridden with the "#" flag). A decimal point appears only if it is followed by a digit (which can be overridden with the "#" flag)

precision The precision specifies the *total* number of significant digits. If the precision is less than 1, then 1 is assumed. The default precision is 6.

flags		
	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it and keep trailing zeros

c The corresponding argument is converted to a character (the value mod 256 is used).

precision *ignored*

flags		
	-	*ignored*
	+	*ignored*
	space	*ignored*
	#	*ignored*

s Unless the "#" flag is used, the corresponding argument must be a string type (or a pointer) and the value is copied to the output as is. C strings and as is (Pascal packed array of char) strings are copied until a null is encountered (for C strings) or the number of characters specified at the precision is reached. Pascal strings may be processed if the type of the argument is a Pascal string. When the "#" flag is used, the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters.

precision The precision specifies the maximum number of characters to output. The default precision is assumed to be infinite. In that case a C and as is strings are output up to but not including a terminating null character and entire Pascal strings are output.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters

P Unless the "#" flag is used, the corresponding argument must be a Pascal string type (or a pointer) and the value is copied to the output as is. When the "#" flag is used, the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters.

You must use an upper-case %P as shown to output a Pascal string type. If you use a lower-case %p argument, the value displayed is output as a pointer type, which is a hexadecimal number optionally preceded by 0X.

precision The precision specifies the maximum number of characters to output. The default precision is assumed to be infinite. In that case the entire Pascal string is output.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters

t The corresponding argument is converted as a function of its type as follows:

a base type u, d, g, p, or s as appropriate to the type with the precision and flags interpreted as a function of these format codes.

non-base type The value(s) are displayed using a pseudo-Pascal type specification format appropriate to the type of the parameter (for example, a RECORD/struct type is displayed using a Pascal-like RECORD notation). The flags control some of the aspects of the formatted output.

The corresponding argument need not specify a value and instead may specify only a type. In this case, the type definition is displayed, again using the same pseudo-Pascal type specification format.

flags	-	display only the type even if corresponding parameter specifies a value. The type is to be displayed exhaustively, in other words, display every type down to its base type.
	+	display only the type even if corresponding parameter specifies a value.
	space	*ignored*
	#	show all values and offsets in hexadecimal

% A single "%" is output; no parameter is used.

precision	*ignored*	
flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	*ignored*

Examples

```
#-- Displays record definition of system type EventRecord.
#-- Notice how EventRecord.Where, which is of type Point, has been
#-- expanded to show its definition too.

printf "%P", ^pstring($910)^
myProg

printf "%P", *(pstring*) $910
myProg

printf "%-t", EventRecord
RECORD
  WHAT: Word;
  MESSAGE: Long;
  WHEN: Long;
  WHERE: RECORD
           CASE Word OF
             (1):
               (RECORD
                  V: Word;
                  H: Word;
                END);

             (2):
               (RECORD
                  VH: ARRAY [(0, 1)] OF
                        Word;
                END);
         END;
  MODIFIERS: Word;
END


#-- Doing the same thing with a target program variable
#-- of type EventRecord. The %t format specifier lets
#-- the Printf display format be controlled by the
#-- type of the variable displayed.

typeof(myEvent)
   EVENTRECORD

printf "%t", myEvent
RECORD
  WHAT: 3;
  MESSAGE: 16686;
  WHEN: 284053;
  WHERE: RECORD
           CASE Word OF
             (1):
               (RECORD
                  V: 272;
                  H: 267;
                END)
             (- - -);
         END;
  MODIFIERS: 2432;
END
```

Proc...End—define a SADE procedure

Syntax

proc *name*[*arg-name*,...]
  commands
end
  or
proc *name* [(*arg-name*,...)]
  commands
end

Description

SADE procedures are delimited by the Proc...End construct. The procedure name is followed by an optional parameter list; if present, the list identifies parameters by name only. Parameters are not assigned a type but instead take on the types of the actual parameter values when the procedure is called.

The parameter list may optionally be enclosed in parentheses. If the parentheses are included in the definition, they must also be used when the procedure is called (and vice versa).

The number of actual parameters need not match the number of formal parameters in the definition. If too few actual parameters are specified, the formal parameters for which there were no corresponding actual parameters are assigned a special undefined value. Extra actual parameters have no corresponding formal name but can be referenced through the predefined SADE variable Arg, which lets you access the parameters of a procedure with references of the form `arg[n]`. The number of the last actual parameter specified is contained in the predefined SADE variable NArgs. Note that the values of these variables represent the parameter state of the currently active procedure and are not defined outside it.

Procedures may be redefined. A procedure must be defined before a call to it can be processed. If you want mutually recursive procedures, a "dummy" procedure (similar to a Pascal FORWARD definition) must first be defined. A second procedure can then redefine the first one, referencing the second procedure. The minimal dummy procedure definition is: `proc foo; end;`.

Procedure calls may be nested.

Example

```
#-- This procedure illustrates the use of the
#-- Arg(n) and Nargs built-in SADE variables

proc myProc (arg1, arg2, arg3, arg4)
define looper
  "in myproc"
  printf( "called with nargs: %d \n", nargs)
  for looper := 1 to nargs
    printf ("arg %d: %d \n", looper, arg[looper])
  end
  "\n"
end

"call myproc with 1 arg"
myproc(1)
"call myproc with 2 args"
myproc(1,2)
"call myproc with 3 args"
myproc(1,2,3)
"call myproc with 4 args"
myproc(1,2,3,4)

#-- output

call myproc with 1 arg
in myproc
called with nargs: 1
arg 1: 1

call myproc with 2 args
in myproc
called with nargs: 2
arg 1: 1
arg 2: 2

call myproc with 3 args
in myproc
called with nargs: 3
arg 1: 1
arg 2: 2
arg 3: 3

call myproc with 4 args
in myproc
called with nargs: 4
arg 1: 1
arg 2: 2
arg 3: 3
arg 4: 4
```

See also Func

Examples

```
redirect "whyNot.§"   # Replace the current selection

#-- In the next example, the Redirect command creates a file to hold
#-- output from SADE. Entering a string echos the string.
#-- That output is redirected to the file, becoming
#-- the file's contents.
#-- Here the string is a comment and a SADE command to execute
#-- the contents of the next file in the chain.

open 'exec1'
redirect 'exec1'
  '"\n executing exec1 now"'
  "execute 'exec2'"

  open "exec2"
  redirect 'exec2'
    "'now executing exec2'"
    "execute 'exec3'"

    open "exec3"
    redirect 'exec3'
      '"Done in exec3"'

redirect pop all
execute "exec1"

Alert "Try a tile windows here\n∂
   Then look at the worksheet for output"

#-- output

executing exec1 now
now executing exec2
Done in exec3
```

Redirect—redirect output

Syntax

redirect [append] *filename*
or
redirect [pop] [all]

Description

The Redirect command redirects the output from SADE commands to the specified file. The simplest way to use redirection is to replace the contents of the named file. If you specify the **append** keyword, the output is appended to the end of the file. You can also use the § character (Option-6) to replace or append to the selection in an open window (see the example below).

▲ **Warning** Be aware that if you use use Redirect to replace the contents of a file that's not open, there's no way to undo it. (If the file is open, you can close it and respond "No" when the dialog asks whether to save changes.) ▲

You can nest Redirect commands to as many as 10 different files; SADE maintains the names of these files as a last-in, first-out queue. If you use the **pop** keyword, or if you use no parameters at all, the output from SADE commands is redirected to the file at the head of the queue. If **all** or **pop all** is specified, standard output is redirected to the current command window.

◆*Note:* Any error conditions cause SADE to perform an implicit **pop all** for any redirected files; this ensures that output returns to the current command window.

Resource—display the resource map

Syntax resource [display] [*addr*] [restype '*type* ']

Description The Resource command displays the contents of your application's resource maps and the system resource map. If you want to display only a particular map, *addr* should be the address of the map. The information displayed for each map includes: its location, the resource ID, the resource type, the value of the master pointer, whether the resource is locked or unlocked, and the resource name. If a resource isn't loaded, the master pointer field says "not loaded."

You can also restrict the display to a particular resource type by using the **restype** keyword with the desired *type*. Note that *type* is case-sensitive and should be enclosed in single quotation marks ('WIND', for example).

Example

```
resource restype 'WIND'
#-- output
Resource Map at $00316EF8
   ResId  RType  MasterPtr  Locked?   Name
    1000  WIND   $00316BE4  Unlocked
    1001  WIND   $00316C08  Unlocked
    1002  WIND   $00316484  Unlocked
    1003  WIND   $003164B8  Unlocked
    1004  WIND   $003164D8  Unlocked
Resource Map at $0002B19C
   ResId  RType  MasterPtr  Locked?   Name
  -16000  WIND   NotLoaded
  -15968  WIND   NotLoaded
  -15840  WIND   NotLoaded
```

See also Resource Check

Repeat...Until—conditionally repeat commands

Syntax

repeat
commands
until *Boolean*

Description

The Repeat ... Until construct provides conditional looping with a test at the end of the loop. The enclosed commands are executed until *Boolean* is true. The enclosed commands are executed at least once.

Repeat constructs may be nested.

Example

```
target 'MyProg'
break \MyProg.MainEventLoop.(1)
launch 'MyProg'
#-- hasEvent is a global variable in MyProg initialized to 0
printf "Pc at %t \nEventReceived? %t \n", where (pc), hasEvent

define dummy
repeat
  step
  dummy := addrtosource(pc, 1)
until hasEvent

printf "\nPc at %t \nEventReceived? %t \n", where (pc), hasEvent

#-- output

Pc at MAINEVENTLOOP.(1)
EventReceived? FALSE

Pc at MAINEVENTLOOP.(32)
EventReceived? TRUE
```

See also Leave

Return—return from a procedure or function

Syntax return [*result*]

Description The Return command returns you from a procedure or function currently in execution. When returning from a function, the function *result* must be specified. (When returning from a procedure, there is no return value.)

Example

```
func MiscTypes (index)
  define global SadeArray[4] := (1, "this is two", 3.3, 4)
  return SadeArray[index]
end

define selector
for selector := 1 to 4 do
  printf "%t \n", MiscTypes(selector)
end

#-- output

1
this is two
3.3
4
```

See also Func, Proc

Resource Check—check the resource map

Syntax resource check [*addr*]

Description The Resource Check command checks the target application's resource maps for consistency. If you want to check a particular map, *addr* should point to the address of the map. If an inconsistency is found, the command displays a diagnostic message specifying the problem.

See also Resource

Save—save a file

Syntax save [all | *filename*]

Description The Save command saves the specified file or, if **all** is specified, saves all files. *Filename* is a string expression and must be enclosed in quotation marks if it's a string constant. If the specified file wasn't modified since the last time it was saved, Save does nothing.

If no parameters are given, Save saves the target window.

Example

```
save 'myFile'
```

See also Open, Close

SADEKey—define a key for entering SADE

Syntax sadekey [*keycode*]

Description The SADEKey command lets you specify a different Command-Option key combination for entering SADE. A complete list of keycodes can be found in the Toolbox Event Manager chapter of *Inside Macintosh* Volume V. To see what key is specified as the SADEKey, just type `sadekey`.

Example

```
sadekey 33      #-- define Command-Option-Delete combination as SADEKey
```

SourcePath—tell SADE where your source files are

Syntax sourcepath [[add | del[ete]] *directoryname* , ...]

Description The SourcePath command tells SADE what directory your source files are in. If your source files are in more than one directory, you can give the SourcePath command a list. If you're unsure which directories you've specified, simply enter `sourcepath` and the current search path is displayed.

If you want to add a directory to, or delete a directory from, the previously specified directories in the search path, you can use the **add** or **delete** keywords. Note that if you specify a directory without using the **add** keyword, any directories previously specified are replaced.

Examples

```
sourcepath 'srcdir', ':myotherdir' # sources in more than one directory
sourcepath add ":samples"          # add directory Samples to search
path
```

See also Directory

Shutdown—shut down or restart the machine

Syntax shutdown [restart]

Description The Shutdown command terminates SADE and calls the Shutdown Manager. If **restart** is specified, the Macintosh is restarted. Be aware that all unsaved work is lost.

Step—single step execution

Syntax step [asm | line] [into]

Description The Step command lets you execute your program one step at a time, from either the source code level or the object code level. If **line** (the default) is specified, execution proceeds one source statement at a time. If the source window containing the current line can be found, the next line to be executed is indicated.

If **asm** is specified, execution proceeds one instruction at a time; the instruction at the program counter is executed and SADE is re-entered. Traps are always treated as single instructions; SADE steps over them, stopping at the first instruction following the trap. Subroutines called by JSR and BSR instructions can either be stepped over or stepped into. If **into** is specified, SADE steps in, stopping at the first instruction of the subroutine. If **into** is omitted, BSRs and JSRs are treated as single instructions.

▲ **Warning** Don't try to step over a routine that does not return to the caller; for instance, a call to `longjmp()`. SADE steps over procedure and function calls by setting the trace bit until after the JSR or BSR is executed, and then replaces the return address with the address of a SADE routine. Since `longjmp()` restores a previously saved register set, including a new stack pointer, SADE's return is lost. If you want to go to the routine restored by longjump0, execute `step asm into` until the registers have been modified and then do a source step. Or better still, if you know where the jump will go, set a breakpoint in that routine. ▲

Example

```
proc stepProc
  "\ncurrent pc"
  disasm pc 4
  "\nstep by instruction twice"
  step asm
  disasm pc 1
  step asm
  disasm pc 1
  "\nstep into a procedure call"
  step into
  disasm pc 4
```

Stack—display stack frames

Syntax stack [*count*] [at *addr*]

Description The Stack command displays a list of the stack frames for the target application. The stack frames displayed are based on register A6 or *addr* if **at** is specified.

For each entry, Stack gives the address of the stack frame, the name of the procedure or function that allocated the frame, and the name and offset (if available) of the parent procedure.

If an explicit *count* is specified, then at most that many stack frames (counting back from the current frame) are displayed.

Example

```
stack at DisplayText.(6)
stack
 Frame Addr  Frame Owner   Called From
  <main>      CMain
  $0032BC24   main          CMain+$0028
  $0032BB2C   SkelMain      main.(51)
  $0032BB0C   LogEvent      SkelMain.(13)+$0012
  $0032BADC   ReportUpdate  LogEvent.(50)+$0004
  $0032BACC   DisplayText   ReportUpdate.(1)+$0004
```

Stop—terminate break action

Syntax stop

Description The Stop command terminates the current break action and returns you to SADE. If the current execution was within a structured statement (Begin...End, for instance), or if multiple commands were selected, the pending commands are executed. To terminate a break action and cancel pending commands, see the Abort command.

Example

```
directory  'VolName:Path:toMyProg:'
launch 'myProg'

proc WhichEvent(stopType)
  define global EventType[16] := ('null',
  'mouse-down','mouse-up',
  'key-down', 'key-up','auto-key',
  'update', 'disk-inserted','activate',
  'network', 'device driver', 'app1',' app2', 'app3', 'app4')

  if theEvent.what  = stopType then
    printf "%P received, stopping\n", EventType[stopType+1]
    stop
  else
    EventType[theEvent.what+1]
    printf
  end
end

break _waitNextEvent from applzone..applzone^ whichEvent(1)
go

#-- output

key-down
update
key-down
mouse-down received, stopping
```

See also Abort, Break, Quit

```
  "\nstep by statement line twice"
  step line
  disasm pc 4
  step
  disasm pc 4

  stop
end

kill 'events'

target 'myProg'
break \myProg.MAINEVENTLOOP.(11) stepProc
launch 'myProg'
go

#-- output

current pc
MAINEVENTLOOP
   +004C  0011F706  486D FFBE       **PEA         -$0042(A5)
   +0050  0011F70A  2F0E             MOVE.L      A6,-(A7)
   +0052  0011F70C  4EBA FE94        JSR           CALLEDPROC    ;
0011F5A2
   +0056  0011F710  42A7             CLR.L       -(A7)

step by instruction twice
MAINEVENTLOOP
   +0050  0011F70A  2F0E            *MOVE.L      A6,-(A7)
MAINEVENTLOOP
   +0052  0011F70C  4EBA FE94       *JSR          CALLEDPROC    ;
0011F5A2

step into a procedure call
CALLEDPROC
   +0000  0011F5A2  4E56 FFFC       *LINK         A6,#$FFFC
   +0004  0011F5A6  2F2D FFE0        MOVE.L      -$0020(A5),-(A7)
   +0008  0011F5AA  A873             _SetPort                          ;
A873
   +000A  0011F5AC  42A7             CLR.L       -(A7)

step by statement line twice
CALLEDPROC
   +0004  0011F5A6  2F2D FFE0       *MOVE.L      -$0020(A5),-(A7)
   +0008  0011F5AA  A873             _SetPort                          ;
A873
   +000A  0011F5AC  42A7             CLR.L       -(A7)
   +000C  0011F5AE  A975             _TickCount                        ;
A975
CALLEDPROC
   +000A  0011F5AC  42A7            *CLR.L       -(A7)
   +000C  0011F5AE  A975             _TickCount                        ;
A975
   +000E  0011F5B0  2D5F FFFC        MOVE.L      (A7)+,-$0004(A6)
   +0012  0011F5B4  42A7             CLR.L       -(A7)
```

Trace—set tracepoints

Syntax

trace *addr*,...
or
trace *trap* [from *addr-range*],...
or
trace *trap-range* [from *addr-range*],...
or
trace all traps [from *addr-range*]

Description

The Trace command sets tracepoints on the specified address or traps within the target program. Tracepoints can be set on a single trap, a range of traps, or on all traps. Traps can be specified by either trap name or trap number. Trap numbers must be prefixed with the "†" character and trap names must be preceded by an underscore.

After setting the tracepoints, you can resume program execution. When the tracepoint is encountered, a message is displayed on standard output, reporting the address or trap being traced, with a symbolic representation of the address if possible. If *addr-range* is specified, the message is displayed only if the trap was called from the specified memory range. In any case, program execution resumes after the message is displayed.

You can specify multiple tracepoints, separated by commas, with a single Trace command.

To remove a tracepoint, use the Untrace command.

Example

```
trace _OpenResFile..._GetResource     #use a trap range
trace †$A997..†$A9A0                  #use a trap range
```

See also

Untrace

Target—tell SADE about your application

Syntax target [*progname* [using *symbolfilename*]]

Description The Target command tells SADE the name of the application you want to debug and identifies the symbol file (the .SYM file generated by the linker). If the .SYM file is already in the same directory as the application and is called progname.SYM (which it usually is), you don't need to bother with the **using** keyword.

Progname and *symbolfilename* are string expressions and must be enclosed in quotes if they are string constants.

To find out what the current target is, just type `target`.

Example

```
target "VolName:MPW:MPW Shell" using "VolName:MPW:ToolStuff:tool.sym"
sourcePath add "VolName:MPW:ToolStuff:"
break \ToolMain.main.(1)
launch "mpw shell"

#-- Run tool to break to SADE with tool as target
#-- and pc at main.(1)
```

Undefine—remove definitions

Syntax undefine *name*,...

Description The Undefine command removes the definition of the specified global SADE variable, procedure, function, or macro. You can supply a list of names to remove multiple definitions. Note that Undefine does not remove local variables defined within SADE procedures or functions.

If you want to redefine an item, you don't need to use Undefine; you can just assign a new value to the existing name using the Define command.

Example

```
proc ControlledProc
  printf "%d   ", ControllerGlobal
end

proc Controller
  define global ControllerGlobal
  define max := 7

  for ControllerGlobal := 1 to max
    ControlledProc
  end
  Undefine ControllerGlobal
  Undefine ControlledProc
end

#-- output

Controller
1   2   3   4   5   6   7

ControllerGlobal
### Could not find "ControllerGlobal" as a program symbol

ControlledProc
### Could not find "ControlledProc" as a program symbol
```

See also Proc, Func, Macro, Define

Unbreak—remove breakpoints

Syntax

unbreak *addr*,...
or
unbreak *trap*,...
or
unbreak *trap-range*,...
or
unbreak all [traps | addrs]

Description

The Unbreak command clears the breakpoint, as well as any associated break action, for the specified addresses or traps. The **all** keyword clears all breaks set in the target program. The **all** keyword can optionally be followed by **traps** or **addrs** to restrict the command to traps or addresses respectively.

Example

```
unbreak _GetResource     #undo break on GetResource trap
```

See also Break

Version—display SADE version information

Syntax version

Description The Version command displays the current SADE version number.

Untrace—remove tracepoints

Syntax

untrace *addr*,...
or
untrace *trap*,...
or
untrace *trap-range*,...
or
untrace all [traps | addrs]

Description

The Untrace command clears the tracepoint at the specified addresses or traps. The **all** keyword clears all tracepoints within the target program. The **all** keyword can optionally be followed by the **traps** or **addrs** keywords to restrict the command to traps or addresses respectively.

Example

```
untrace _GetResource     #undo trace on _GetResource trap
```

See also Trace

Index

While...End—conditionally repeat commands

Syntax

while *Boolean* [do]
  commands
end

Description

The While...End construct provides conditional looping with a test at the beginning of the loop. The enclosed commands are executed as long as *Boolean* is true. If the condition is false at the outset, the enclosed commands are never executed.

While constructs may be nested.

Example

```
define goSmall := 10
while goSmall > -2 do
  while goSmall > 4 do
    while goSmall > 7 do
      printf "          Inner loop - goSmall = %d\n", goSmall
      goSmall := goSmall -1
    end
    printf "     Middle loop - goSmall = %d\n", goSmall
    goSmall := goSmall - 1
  end
  printf "Outer loop - goSmall = %d\n", goSmall
  goSmall := goSmall - 1
end
#-- output
          Inner loop - goSmall = 10
          Inner loop - goSmall = 9
          Inner loop - goSmall = 8
     Middle loop - goSmall = 7
     Middle loop - goSmall = 6
     Middle loop - goSmall = 5
Outer loop - goSmall = 4
Outer loop - goSmall = 3
Outer loop - goSmall = 2
Outer loop - goSmall = 1
Outer loop - goSmall = 0
```

See also Leave

THE APPLE PUBLISHING SYSTEM

This Apple® manual was written, edited, and composed on a desktop publishing system using Apple® Macintosh® computers and Microsoft® Word software. Proof and final pages were created on the Apple LaserWriter® IINTX printer. POSTSCRIPT®, the LaserWriter® page-description language was developed by Adobe Systems Incorporated. The illustrations were created using Adobe Illustrator and some were output to a Linotronic 300.

The illustration on the cover was generated using Adobe Illustrator 88 on a Macintosh® II computer. Some of the images were scanned using an Apple® Scanner and then manipulated in ImageStudio. Initial proofing was done using a QMS color printer. Color separations were done using Adobe separator and output to a Linotronic 300 at standard resolution.

Text type is Apple's corporate font, a condensed version of Garamond. Bullets are ITC Zapf Dingbats®. Some elements, such as programs listings, are set in Apple Courier, a fixed-width font.